EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)

Volume 41 (2022), *Number 2*

Supplemental Material for “Dressi: A Hardware-Agnostic Differentiable Renderer with Reactive Shader Packing and Soft Rasterization”

Yusuke Takimoto*, Hiroyuki Sato*, Hikari Takehara*, Keishiro Uragaki,
Takehiro Tawara, Xiao Liang, Kentaro Oku, Wataru Kishimoto, and Bo Zheng

Huawei Technologies Japan K.K.

This supplemental document describes implementation details and additional applications.

1. Implementation Details

To clarify the implementation of our proposed system, we describe class structures, API, and algorithm details in C++ code style. Constant, reference, pointer, parallelization, and other less important parts are omitted for readability.

1.1. Example of the Optimization on Dressi Renderer

The following program is an example of the vertex positions and normal map optimization on Dressi renderer with HardSoftRas and physically-based shading. Although Dressi code for optimization looks similar to PyTorch, Dressi is define-and-run. At first, a renderer instance is declared (details in Section 1.2) and scene files are loaded. Second, a forward computational graph is built for rendering and loss calculation. Third, the vertex positions and normal map are specified as optimization targets by the requires-gradient flag as in PyTorch. After these setups, a main optimization process is executed using `execStep()`. Finally, the optimized results are saved.

```
int main(int argc, char *argv[]) {
  OptimizeVertexPostionAndNormalTexture();
  return 0;
}

void OptimizeVertexPostionAndNormalTexture() {
  // Create renderer instance
  DressiBasicRenderer renderer;
  // Load scene data on CPU
  renderer.loadScene("initial_scene.gltf", "target_img.png");

  // Build a computational graph
  uint32_t K = 2;         // The number of peeling for HardSoftRas
  float r = 0.01f;        // Radius parameter for HardSoftRas
  float sigma = r / 7.f;  // Blending parameter for HardSoftRas
  float delta = r;        // Silhouette edge width for HardSoftRas
  float lr = 0.01f;       // Learning rate for optimizer
  renderer.buildGraph(K, r, sigma, delta, lr);
  // Mark vertex positions and a normal texture as optimization targets
  renderer.setRequiresGrad("vtx_pos");
  renderer.setRequiresGrad("normal");

  // Optimization interations
  for (int iter = 0; iter < 1000; iter++) {
    renderer.execStep();
  }

  // Save optimized data as a GLTF file
  renderer.saveScene("optimized_scene.gltf");
}
```

* These authors contributed equally to this work

1.2. Example of the Dressi Renderer

`DressiBasicRenderer` is an example of rendering and optimization based on Dressi system design. It has a `DressiAD` instance, which is DR-specialized AD, and the renderer considers only the forward pass and optimizers (e.g., SGD). The implementation details of `DressiAD` are described in Section 1.6. Scene data loaded to the CPU memory are parsed as `CpuImage` structure, which is a simple buffer of a 2D image. Non-image data (e.g., vertex positions) are converted into 2D image representation for efficient rendering. After `CpuImages` are loaded, `Variable` objects are created to represent a computational graph. It is a data structure for inputs and outputs of functions (details in Section 1.4). Forward rendering and optimizer process are described by assembling operators and namespace `F` functions (i.e., `F::`) for the `Variable` objects, constructing a computational graph. The operators and `F::` functions wrap `Function` objects, which have a forward GLSL code and backward generator. This implementation direction is similar to common AD libraries such as PyTorch and TensorFlow. Examples of the `F::` functions are described in Section 1.5, and the implementation details of `Function` are described in Section 1.4. `BuildBasicRenderGraph()` is an example of rendering functions using HardSoftRas (details in Section 1.3). The loss for the rendered image and an optimizer algorithm are set to the `DressiAD` instance.

```
// DressiBasicRenderer: An example of renderer class for Dressi system.
class DressiBasicRenderer {
public:
  void loadScene(std::string gltf_filename,
                 std::string target_img_filename) {
    // Load initial scene from a file and parse to CpuImage structures
    std::tie(m_img_map["vtx_pos"], m_img_map["vtx_uv"], m_img_map["faces"],
             m_img_map["world_mat"], m_img_map["view_mat"],
             m_img_map["prj_mat"], m_img_map["env_img"],
             m_img_map["albedo"], m_img_map["metallic"],
             m_img_map["roughness"], m_img_map["normal"],
             m_img_map["background"]) = LoadGltfAsCpuImages(gltf_filename);
    // Load target an image file and parse to a CpuImage structure
    m_img_map["target"] = LoadTargetImageAsCpuImage(target_img_filename);

    // Create top variables of a computational graph
    m_var_map["vtx_pos"] = {VEC3, m_img_map["vtx_pos"].getImgSize()};
    m_var_map["vtx_uv"] = {VEC2, m_img_map["vtx_uv"].getImgSize()};
    m_var_map["faces"] = {IVEC3, m_img_map["faces"].getImgSize()};
    m_var_map["model_mat"] = {MAT4, {1, 1}};
    m_var_map["view_mat"] = {MAT4, {1, 1}};
    m_var_map["prj_mat"] = {MAT4, {1, 1}};
    m_var_map["env_img"] = {VEC3, m_img_map["env_img"].getImgSize()};
    m_var_map["albedo"] = {VEC3, m_img_map["albedo"].getImgSize()};
    m_var_map["metallic"] = {FLOAT, m_img_map["metallic"].getImgSize()};
    m_var_map["roughness"] = {FLOAT, m_img_map["roughness"].getImgSize()};
    m_var_map["normal"] = {VEC3, m_img_map["normal"].getImgSize()};
    m_var_map["background"] = {VEC3, m_img_map["background"].getImgSize()};
    m_var_map["target"] = {VEC4, m_img_map["target"].getImgSize()};

    // Set a flag to send CPU images to GPU
    m_is_sent = false;
  }

  void buildGraph(uint32_t K, float r, float sigma, float delta,
                  float lr) {
    // Build a rendering computational graph
    Variable rendered_img = BuildBasicRenderGraph(
            m_var_map["vtx_pos"], m_var_map["vtx_uv"], m_var_map["faces"],
            m_var_map["model_mat"], m_var_map["view_mat"],
            m_var_map["prj_mat"], m_var_map["env_img"],
            m_var_map["albedo"], m_var_map["metallic"],
            m_var_map["roughness"], m_var_map["normal"],
            m_var_map["background"], K, r, sigma, delta);
    // Take L1 loss
    Variable loss = F::Mean(F::Abs(m_var_map["target"] - rendered_img));

    // Set the loss and an optimizer to DressiAD
    m_dressi_ad.setLossVar(loss);
    m_dressi_ad.setOptimizer([=](Variables xs, Variables gxs) {
        // SGD for all inputs
        Variables updated_xs;
        for (size_t i = 0; i < xs.size(); i++) {
          updated_xs.push_back(xs[i] - gxs[i] * lr);
        }
        return updated_xs;
    });
  }

  void setRequiresGrad(std::string name) {
    // Set a requires-gradient flag
    m_var_map[name].setRequiresGradRecursively();
  }

  void execStep() {
    // Send CPU images to GPU if needed.
    if (!m_is_sent) {
      for (auto [name, var]: m_var_map) {
        m_dressi_ad.sendImg(var, m_img_map[name]);
      }
      m_is_sent = false;
    }

    // Execute one iteration of the optimization
    m_dressi_ad.execStep();
  }

  void saveScene(std::string gltf_filename) {
    // Receive all GPU images
    for (auto [name, var]: m_var_map) {
      m_img_map[name] = m_dressi_ad.recv(var);
    }
    // Save the optimized scene to a GLTF file.
    SaveGltfFromCpuImages(m_img_map);
  }

private:
  // DressiAD instance for DR-specialized AD
  DressiAD m_dressi_ad;
  // Scene data map for CpuImage
  std::map<std::string, CpuImage> m_img_map;
  // Scene data map for Variable corresponding to CpuImage
  std::map<std::string, Variable> m_var_map;
  // Internal flags
  bool m_is_sent = false;
};
```

1.3. Example of the Rendering Algorithm Using HardSoftRas

`BuildBasicRenderGraph()` describes a rendering example using the HardSoftRas algorithm. All our DR algorithms are written in AD, and there are no special backward declarations. `BuildRasterize()`, which contains Enlarge and Shift operations and depth peeling, rasterizes triangles. `BuildBlend()` takes the outputs of `BuildRasterize()` and shading parameters, and it performs blending operations for color and silhouette images.

```
Variable BuildBasicRenderGraph(Variable vtx_pos, Variable vtx_uv,
        Variable faces, Variables model_mat, Variable view_mat,
        Variable prj_mat, Variable env_img, Variable albedo,
        Variable metallic, Variable roughness, Variable normal,
        Variable background, uint32_t K, float r, float sigma,
        float delta) {
  // Irradiance map
  Variable irrad_img = BuildIrradianceSample(env_img);
  // Pre-filtered environment map for glossy material
  Variable pref_img = BuildPrefEnvironmentSample(env_img)
  // BRDF integration map
  Variable brdf_img = BuildBrdfIntegrationMap();

  /******** The beginning of HardSoftRas algorithm *******/
  // Rasterize and shade for each peeling plane
  auto prev_prj_depth = F::Float(0.f);
  Variables shaded_imgs, stencils, edge_dists;
  for (uint32_t plane_idx = 0; plane_idx < K; plane_idx++) {
    // Rasterize a single geometry
    auto [stencil, edge_dist, world_pos, world_nor, uv, prj_depth] =
              BuildRasterize(vtx_pos, vtx_uv, faces, model_mat,
                             view_mat, prj_mat, prev_prj_depth, r);
    prev_prj_depth = prj_depth;   // Update peeling depth
    // Note: To rasterize multiple geometries, extra depth test is needed

    // Physically-based shading
    auto shaded_img = BuildPBS(stencil, world_pos, world_nor, uv,
                               albedo, metallic, roughness, normal,
                               irrad_img, pref_img, brdf_img);

    shaded_imgs.push_back(shaded_img);
    stencils.push_back(stencil);
    edge_dists.push_back(edge_dist);
  }

  // Blend shaded planes
  auto [blended_shaded_img, blended_silhouette_img] = BuildBlend(
          shaded_imgs, stencils, edge_dists, background, K, sigma, delta);
  /********* The end of HardSoftRas algorithm ********/

  // Apply tone-map and gamma correction
  blended_shaded_img = BuildToneMap(blended_shaded_img);

  // Join silhouette image as alpha
  auto rendered_img = F::Vec4(blended_shaded_img, blended_silhouette_img);

  return rendered_img;
}

auto BuildRasterize(Variable vtx_pos, Variable vtx_uv,
                    Variable faces, Variable model_mat, Variable view_mat,
                    Variable prj_mat, Variable prev_prj_depth, float r) {
  auto vtx_pos = F::Vec4(vtx_pos, 1.f);  // Cast to homogeneous coordinate
  vtx_pos = model_mat * vtx_pos;          // Apply model matrix

  // Compute vertex normals from positions
  auto vtx_nor = BuildNormalCompute(vtx_pos);
  // Apply view and projection matices
  auto vtx_prj = prj_mat * view_mat * vtx_pos;

  // Lookup the vertex buffer with faces to create per-triangle attributes
  auto tri_prj_0, tri_prj_1, tri_prj_2 = LookupFaces(vtx_prj, faces);
  auto tri_pos_0, tri_pos_1, tri_pos_2 = LookupFaces(vtx_pos, faces);
  auto tri_nor_0, tri_nor_1, tri_nor_2 = LookupFaces(vtx_nor, faces);
  auto tri_uv_0, tri_uv_1, tri_uv_2 = LookupFaces(vtx_uv, faces);

  // Enlarge(): Enlarge triangle vertex positions by scaling 'r'
  // Notice that the number of triangles will change, and the original
  // vertex indices are stored.
  auto [large_prj_0, large_prj_1, large_prj_2, original_vtx_idxs] =
          BuildEnlarge(tri_prj_0, tri_prj_1, tri_prj_2, r);

  // Join triangle vertex positions to rasterize the single vertex buffer
  auto large_prj_flat = F::Stack(large_prj_0, large_prj_1, large_prj_2);
  // Rasterize enlarged triangles, and obtain screen space vertex indices
  auto screen_vtx_idxs = F::Rasterize(large_prj_flat, original_vtx_idxs);

  // SignedDist(): Compute pixel-to-edge distance inside enlarged triangles
  auto edge_dist = BuildEdgeDistance(tri_prj_0, tri_prj_1, tri_prj_2,
                                     screen_vtx_idxs);

  // Compute screen space barycentric coordinates
  auto bary_coords = BuildBaryCoord(tri_prj_0, tri_prj_1, tri_prj_2,
                                    screen_vtx_idxs);
  // Interpolate vertex attributes to be screen space
  auto prj_pos = BuildInterpolate(tri_prj_0, tri_prj_1, tri_prj_2,
                                  screen_vtx_idxs, bary_coords);
  auto world_pos = BuildInterpolate(tri_pos_0, tri_pos_1, tri_pos_2,
                                    screen_vtx_idxs, bary_coords);
  auto world_nor = BuildInterpolate(tri_nor_0, tri_nor_1, tri_nor_2,
                                    screen_vtx_idxs, bary_coords);
  auto uv = BuildInterpolate(tri_uv_0, tri_uv_1, tri_uv_2,
                             screen_vtx_idxs, bary_coords);

  // Shift(): Depth modification
  auto prj_depth = F::GetW(prj_pos);
  auto soft_depth = edge_dist * 0.5f + 0.5f;
  auto hard_depth = prj_depth * 0.5f;
  auto is_inside = (0.f < edge_dist);
  prj_depth = F::Mix(soft_depth, hard_depth, is_inside);
  prj_depth = F::SetFragDepth(prj_depth);  // Set depth to gl_FragDepth
  // Depth peeling
  auto stencil = F::PeelDepth(prj_depth, prev_prj_depth);

  return {stencil, edge_dist, world_pos, world_nor, uv, prj_depth};
}

auto BuildBlend(Variables shaded_imgs, Variables stencils,
                Variables edge_dists, Variable background, uint32_t K,
                float sigma, float delta) {
  // Compute pixel weights from edge distances
  Variables weights;
  Variables weighted_cols;
  for (uint32_t plane_idx = 0; plane_idx < K; plane_idx++) {
    auto prob = 1.f / (1.f + F::Exp(edge_dists[i] / -sigma));
    auto weight = prob * stencils[plane_idx];
    weights.push_back(weight);
    weighted_cols.push_back(weight * shaded_imgs[plane_idx]);
  }

  // Blend shaded color by normalized weights
  auto sum_stencils = F::SumPixelWise(stencils);
  auto sum_weight = F::SumPixelWise(weights) / sum_stencils;
  auto sum_weighted_col = F::SumPixelWise(weighted_cols) / sum_stencils;
  auto blended_col = F::Mix(background, sum_weight, sum_weighted_col);

  // Blend silhouettes
  auto blended_sil = F::Float(1.f);
  for (uint32_t plane_idx = 0; plane_idx < K; plane_idx++) {
    blended_sil *= (1.f - weights[plane_idx]);
  }
  blended_sil = 1.f - blended_sil;

  // Blend hard faces by an edge mask
  auto is_hard = (0.f < edge_dists[0]);
  auto hard_shaded = shaded_imgs[0] * is_hard;  // Mask inside
  auto edge_mask = BuildEdgeMask(edge_dists[0], delta);
  blended_col = F::Mix(hard_shaded, blended_col, edge_mask);
  blended_sil = F::Mix(is_hard, blended_sil, edge_mask);

  return {blended_col, blended_sil};
}
```

1.4. Function and Variable Objects

`Functions` and `Variables` are the basic structures for representing the graph structure of a computational flow. They have cross-references for constructing computational graphs. `Function` has a GLSL snippet for its forward process and a generator function for the backward pass.

```
// Variable Data Types
enum VType { FLOAT, VEC2, ..., MAT2, ..., INT, IVEC2, ... };

// Variable Object
class Variable {
public:
  Variable(VType vtype = FLOAT, ImgSize img_size = {1, 1}):
      m_vtype(vtype), m_img_size(img_size) {}

  // Getters / setters
  VType getVType() { ... }
  void setVType(VType vtype) { ... }
  ImgSize getImgSize() { ... }
  void setImgSize(ImgSize img_size) { ... }
  Function getCreator() { ... }
  void setCreator(Function creator) { ... }
  std::vector<Function> getUsers() { ... }
  void addUser(Function user) { ... }
  bool getRequiresGrad() { ... }
  void setRequiresGrad(bool req_grad) { ... }
  void setRequiresGradRecursively(bool req_grad) { ... }
  bool IsDirty() { ... }
  void setDirty(bool is_dirty) { ... }
  void setDirtyRecursively(bool is_dirty) { ... }

private:
  VType m_vtype;
  ImgSize m_img_size;
  Function m_creator = Empty;
  std::vector<Function> m_users;
  bool m_req_grad = false;
  bool m_is_dirty = true;
};

// Aliases for simplification
using Variables = std::vector<Variable>;
using BwdFunc = std::function<Variable(Variables xs, Variable y,
                                       Variable gy, uint32_t bwd_idx)>;
// Shader type
enum ShaderType { FRAG, COMP, RASTER };

// Function Object
class Function {
public:
  Function(std::string code, ShaderType type, BwdFunc bwd_func) :
    m_fwd_code(code), m_type(type), m_bwd_func(bwd_func) {}

  // Getters
  std::string getFwdCode() { ... }
  ShaderType getShaderType() { ... }
  Variables getInputVars() { ... }
  Variable getOutputVar() { ... }

  // Building forward/backward connections
  Variable buildFwd(Variables xs) {
    m_xs = xs;
    for (auto x: xs) { x.addUser(this); }
    m_y.setCreator(this);
    // Infer and set variable type and image size of output
    m_y.setVType(InferOutputVType(xs));
    m_y.setImgSize(InferOutputImgSize(xs));
    return m_y;
  }
  Variable buildBwd(Variable gy, uint32_t bwd_idx) {
    // Create the graph for backward computation
    return m_bwd_func(m_xs, m_y, gy, bwd_idx);
  }

private:
  std::string m_fwd_code;
  ShaderType m_type;
  BwdFunc m_bwd_func;
  Variables m_xs;
  Variable m_y;
};

// Alias for simplification
using Functions = std::vector<Function>;
```

1.5. Examples of the Namespace F Functions

Regarding Function and Variable objects, concrete operators/functions such as addition and multiplication are defined in namespace `F`. Similar to common AD libraries, both forward native code and backward generation are described in `F::` functions. In contrast to common AD libraries, DR-specific functions such as `F::Rasterize()` and `F::PeelDepth()` are also included in our AD as no-backward functions. Each function declaration contains a GLSL snippet (e.g. `{y}={x0}+{x1};`). `{x0}` and `{x1}` represent the first and second inputs of the function, and `{y}` indicates an output. The snippet is used to generate a fragment or compute shader code in Section 1.8. The input and output markers can be replaced by actual variable names. Most functions can be written as shader snippets, but rasterization cannot be written as such. Therefore, `F::Rasterize` function is specially marked as `ShaderType::RASTER`.

```
// Function Operators
namespace F {

Variable Float(float fv) {  // Constant float in GLSL
  return Function("{y}=float(" + fv + ");", FRAG, [](xs, y, gy, bwd_idx) {
    return nullptr;  // No backward
  }).buildFwd({});
}
Variable Add(Variable x0, Variable x1) {
  return Function("{y}={x0}+{x1};", FRAG, [](xs, y, gy, bwd_idx) {
    return gy;
  }).buildFwd({x0, x1});
}

Variable Mul(Variable x0, Variable x1) {
  return Function("{y}={x0}*{x1};", FRAG, [](xs, y, gy, bwd_idx) {
    if (bwd_idx == 0) {
      return gy * xs[1];  // Backward pass toward 'x0'
    } else {
      return gy * xs[0];  // Backward pass toward 'x1'
    }
  }).buildFwd({x0, x1});
}

Variable Sin(Variable x) {
  return Function("{y}=sin({x0});", FRAG, [](xs, y, gy, bwd_idx) {
    return F::Cos(gy);
  }).buildFwd({x});
}

Variable Rasterize(Variable vtx_pos, Variable vtx_attrib) {
  return Function("{y}={x1};",  // Rasterizing an attribute 'vtx_attrib'
        RASTER,                 // Marked as rasterization specially
        [](xs, y, gy, bwd_idx) { return nullptr; }  // No backward pass
  ).buildFwd({x});
}

Variable PeelDepth(Variable frag_depth, Variable prev_frag_depth) {
  return Function("if({x0}<={x1})discard; {y}=1.0;", FRAG,
        [](xs, y, gy, bwd_idx) { return nullptr; }  // No backward pass
  ).buildFwd({frag_depth, prev_frag_depth});
}

...
}
```

1.6. Dressi-AD

The Dressi-AD class has interfaces to set loss variables and an optimizer (`setLossVar()` and `setOptimizer()`, respectively), transfer data between CPU and GPU (`sendImg()` and `recvImg()`, respectively), and execute optimization iteration (`execStep()`). `execStep()` contains setups for the backward pass construction (Section 1.7), optimizer function construction, computational graph traversal, substage packing (Section 1.8), stage packing (Section 1.9), Vulkan object creation (Section 1.10) from stage and substage graphs, and Vulkan command execution.

```
// DressiAD: DR-specialized AD library class
class DressiAD {
public:

  // Optimizer function
  //  (takes inputs and thier gradients, and returns optimized outputs.)
  using Optimizer = std::function<Variables(Variables xs, Variables gxs)>;

  // Setter
  void setLossVar(Variable loss_var) {
    m_loss_var = loss_var;
    m_init_status = InitStatus::BACKWARD; // Execute initialize process
  }
  void setOptimizer(Optimizer optim_func) { ... }

  // Building status
  enum InitStatus { BACKWARD, OPTIMIZER, TRAVERSE, SUBSTAGE, STAGE,
                    VULKAN, FINISHED};

  // Execute one step of rendering and optimization
  void execStep() {
    // Check dirty flags of input variables
    if (IsAnyVariableDirtyChanged(m_input_vars)) {
      m_graph_static_cnt = 0;
    }
    // Check rebuild condition
    if (m_graph_static_cnt == FAST_REBUILD_COUNT) {
      m_init_status = InitStatus::STAGE; // Execute fast rebuild
    } else if (m_graph_static_cnt == FULL_REBUILD_COUNT) {
      m_init_status = InitStatus::SUBSTAGE; // Execute full rebuild
    }

    if (m_init_status <= InitStatus::BACKWARD) {
      // 1) Traverse the forward computational graph and
      //    generate the backward graph by Function::buildBwd()
      m_input_vars, m_input_grad_vars = BuildBackward(m_loss_var);
    }
    if (m_init_status <= InitStatus::OPTIMIZER) {
      // 2) Build the optimizer
      //    to connect the forward pass with the backward pass
      m_updated_vars = m_optim_func(m_input_vars, m_input_grad_vars);
      m_upd_inp_map = CreateMap(m_input_vars, m_updated_vars);
    }
    if (m_init_status <= InitStatus::TRAVERSE) {
      // 3) Traverse full computational graph
      //    from updated to input variables
      m_all_funcs = TraverseFuncs(m_updated_vars, m_input_vars);
    }
    if (m_init_status <= InitStatus::SUBSTAGE) {
      // 4) Pack the computational graph into SubStages
      m_substages = PackDirtyFuncsIntoSubStages(m_all_funcs, m_vk_imgs);
    }
    if (m_init_status <= InitStatus::STAGE) {
      // 5) Pack the SubStages into Stages
      m_stages = PackDirtySubStagesIntoStages(m_substages, m_vk_imgs);
    }
    if (m_init_status <= InitStatus::VULKAN) {
      // 6) Parse to Vulkan objects
      m_vk_imgs, m_vk_cmd_buf, m_vk_renderpasses, m_vk_pipelines, ... =
          ParseStagesAsVulkanObjects(m_stages, m_vk_imgs, m_upd_inp_map);
    }

    // 7) Execute a command buffer
    VkQueueSubmit(m_vk_cmd_buf);

    // Mark as clean recursively
    for (auto v: m_input_vars) { v.setDirtyRecursively(false); }
    m_init_status = FINISHED;
  }

  // Transfer image data between CPU and GPU.
  void sendImg(Variable var, CpuImage cpu_img) {
    if (m_vk_imgs.contains(var)) {
      var.setDirty(true);  // Mark as changed
      SendHostImageToDevice(m_vk_imgs[var], cpu_img);  // CPU -> GPU
    }
  }
  CpuImage recvImg(Variable var) {
    if (m_vk_imgs.contains(var)) {
      return ReceiveHostImageFromDevice(m_vk_imgs[var]);  // GPU -> CPU
    }
    return nullptr;
  }

private:
  InitStatus m_init_status = InitStatus::BACKWARD;
  Variable m_loss_var;   // Last variable of the forward pass
  Optimizer m_optim_func;
  uint32_t m_graph_static_cnt = 0;

  Variables m_input_vars;      // Top variables of the forward pass
  Variables m_input_grad_vars; // Last variables of the backward pass
  Variables m_updated_vars;    // Last variables of the computational graph
  Functions m_all_funcs;       // All functions of the computational graph
  std::map<Variable, Variable> m_upd_inp_map;  // Map from input to updated

  SubStages m_substages;
  Stages m_stages;

  std::map<Variable, VkImage> m_vk_imgs;
  VkCommandBuffer m_vk_cmd_buf;
  std::vector<VkRenderPass> m_vk_renderpasses;
  std::vector<VkPipeline> m_vk_pipelines;
  ... // Many Vulkan objects
};
```

1.7. Backward Pass Construction

Backward pass construction is mostly similar to common AD libraries. `BuildBackward()` backward traverses a forward computational graph from loss variables, sums up gradient variables by chain rule, and returns a pair of input and corresponding gradient variables.

```
std::tuple<Variables, Variables> BuildBackward(Variable loss_var) {
  // Mapping from a forward variable to backward gradient variables
  std::map<Variable, Variables> fwd_bwds_map;
  // Function queue that keeps the order of use
  std::priority_queue<Function> func_queue;

  // Set loss as the starting point
  fwd_bwds_map[loss_var] = F::Float(1.f);
  func_queue.push(loss_var.getCreator());
  // Traversal loop
  while (!func_queue.empty()) {
    // Pop the latest-used function.
    Function func = func_queue.top();
    func_queue.pop()
    if (IsSeenFunc(func)) continue;

    // Sum up gradients.
    Variable y = func.getOutputVar();
    Variables gys = fwd_bwds_map.at(y);
    Variable gy = F::SumPixelWise(gys);
    fwd_bwds_map.erase(y);

    Variables xs = func.getInputVars();
    for (size_t x_idx = 0; x_idx < xs.size(); x_idx++) {
      auto x = xs[x_idx];
      if (!x.getRequiresGrad()) continue;  // Skip no gradient path

      // Build a backward connection
      Variable gx = func.buildBwd(gy, x_idx);
      if (!gy) {
        continue;  // Skip non-backwardable function
      }
      // Register a new gradient
      fwd_bwds_map[x].push_back(gx);

      // Push a creator function for recurrent traversal
      Function x_creator = x.getCreator();
      if (x_creator) func_queue.push(x_creator);
    }
  }

  // Collect input and gradient of input variables
  Variables input_vars, input_grad_vars;
  for (auto [x, gxs]: fwd_bwds_map) {
    input_vars.push_back(x);
    input_grad_vars.push_back(F::SumPixelWise(gxs));
  }
  return {input_vars, input_grad_vars};
}
```

1.8. Substage Packing

In substage packing, function objects are packed into substages, whereas clean and cached ones are skipped to reduce the computational cost. "Clean" indicates that the output Variable of a function is marked as clean. "Cached" indicates that the Variable is used as a substage I/O, and its data is stored on GPU as a result of previous iteration. In our implementation, packing starts from the last output of the computational graph using a greedy algorithm. The active substage being currently processed is iteratively updated. At each iteration, the function that has more edges is packed into the active substage under Vulkan constraints in a greedy manner. If there are no packable functions into the active substage, the substage is switched to a new one.

```
struct SubStage {
  Variables vtx_vars; // Vertex buffer inputs for rasterization
  Variables inp_vars; // Input attachment
  Variables tex_vars; // Texture sampler inputs
  Variables slt_vars; // Sampler-less texture inputs
  Variables uif_vars; // Uniform inputs
  Variables out_vars; // Color attachment (outputs)
  Variables gen_vars; // All generated variables including shader inside
  Functions funcs;
  std::string shader_code;
};
using SubStages = std::vector<SubStage>;

SubStages PackDirtyFuncsIntoSubStages(
      Functions all_funcs, std::map<Variable, VkImage> cached_imgs) {
  // Traverse functions, ignoring clean and cached branches.
  auto dirty_funcs = RemoveCleanFuncs(all_funcs, cached_imgs);

  // Graph optimization
  OmitConstantFuncs(dirty_funcs);   // Precompute constant values
  OmitDuplicatedFuncs(dirty_funcs); // Omit same functions with same inputs

  // Search suitable packing under Vulkan limitations
  SubStages substages = SearchSuitableFunctionPacking(dirty_funcs);

  for (auto substage: substages) {
    // Generate GLSL shader codes by string manipulation as following.
    //  1) Collect GLSL snippets of functions in a substage.
    //      ex.) "{y}={x0}+{x1};", "{y}=sin({x0});"
    //  2) Join snippet lines, and replace input and output markers.
    //      ex.) "v7=v5+v6; v8=sin(v7);"
    //  3) Add variable declaration, I/O codes of input attachments,
    //     main function, and description of attachments/uniforms/etc...
    //      ex.) "layout(...) uniform subpassInput sub_inp[2]; ...
    //            void main() { float v5=subpassLoad(sub_inp[0]); ...
    //                          float v7=v5+v6; float v8=sin(v7); ... }"
    substage.shader_code = GenerateShaderCode(substage);
  }
  return substages;
}

SubStages SearchSuitableFunctionPacking(Functions dirty_funcs) {
  SubStages substages;
  SubStage active_substage;
  Variables used_vars;

  while (!dirty_funcs.empty()) {
    // Collect functions whose outputs are not used in other functions.
    Functions candidate_funcs = CollectLatestFuncs(dirty_funcs);
    // Sort functions by edge numbers to the substage.
    // We assume that the number of edges correlates a probability to be
    // a better choice to maximize the size of substage.
    candidate_funcs = SortByEdgeNumbers(candidate_funcs, active_substage);

    // Try packing from the last of computational graph.
    bool is_found = false;
    for (auto func: candidate_funcs) {
      // Try push a function into the substage
      auto trial_substage = PushFrontFuncIntoSubStage(
              func, active_substage, used_vars);
      if (IsSubStageVkLimitsSatisfied(trial_substage)) {
        active_substage = trial_substage;  // Suitable packing found
        dirty_funcs.erase(func);
        is_found = true;
        break;
      }
    }
    if (!is_found) {
      // Switch to a next substage if no more packing
      substages.push_back(active_substage);
      active_substage = {};  // Clear
      // Mask input variables as used for substage dependency
      for (auto inp_var: CollectAllInputVars(substage)) {
        used_vars.push_back(inp_var);
      }
    }
  }
  return substages;
}

SubStage PushFrontFuncIntoSubStage(Function func, SubStage substage,
                                   Variables used_vars) {
  // Register function inputs
  for (auto inp_var: func.getInputVars()) {
    if (func.getShaderType() == RASTER) {
      substage.vtx_vars.push_back(inp_var);  // As vertex buffer
    } else if (IsSamplerType(inp_var)) {
      substage.tex_vars.push_back(inp_var);  // As texture sampler
    } else if (IsSamplerLessType(inp_var)) {
      substage.slt_vars.push_back(inp_var);  // As sampler-less texture
    } else if (inp_var.getImgSize() == {1, 1}) {
      substage.uif_vars.push_back(inp_var);  // As uniform
    } else {
      substage.inp_vars.push_back(inp_var);  // As input attachment
    }
  }

  // Remove generated variables from inputs (Vertex, texture,
  // and sampler-less texture must not be generated in the same substage.)
  Variables out_var = func.getOutputVar();
  substage.inp_vars.erase(out_var);
  substage.uif_vars.erase(out_var);

  // Register function output which is needed by other substages
  if (used_vars.contains(out_var)) {
    substage.out_vars.push_back(out_var);
  }
  // Register function output as generated variables
  substage.gen_vars.push_back(out_var);
  // Register function
  substage.funcs.push_back(func);
  return substage;
}

bool IsSubStageVkLimitsSatisfied(SubStage substage) {
  // All output images must have same image size.
  if (!AreSameImgSizes(substage.out_vars)) return false;
  // All functions must have same shader type except for top rasterization.
  if (!AreSameShaderTypes(substage.funcs) &&
      !(substage.funcs[0].getShaderType() == RASTER &&
        AreSameShaderTypes(RemoveFirst(substage.funcs)))) return false;
  // Vertex/texture input must come from another substage.
  if (substage.gen_vars.containsAny(substage.vtx_vars)) return false;
  if (substage.gen_vars.containsAny(substage.tex_vars)) return false;
  // Limited numbers of Vulkan I/O
  if (MAX_VULKAN_INPUT_ATTACH < substage.inp_vars.size()) return false;
  if (MAX_VULKAN_TEXTURE_SAMPLER < substage.tex_vars.size()) return false;
  if (MAX_VULKAN_SAMPLED_IMAGE < substage.slt_vars.size()) return false;
  if (MAX_VULKAN_UNIFORM < substage.uif_vars.size()) return false;
  if (MAX_VULKAN_OUTPUT_ATTACH < substage.out_vars.size()) return false;
  ...  // Other number limitations for combined conditions
  return true;
}
```

1.9. Stage Packing

Stage packing, which packs substages into stages, follows the same strategy as the substage packing.

```
struct Stage
{
  // I/O variables as same as SubStages'
  Variables vtx_vars, inp_vars, tex_vars, slt_vars, uif_vars, out_vars;
  // Hierarchical structure for SubStages
  SubStages substages;
};
using Stages = std::vector<Stage>;

Stages PackDirtySubStagesIntoStages(
    SubStages all_substages, std::map<Variable, VkImage> cached_imgs)
{
  // Traverse a substage graph, ignoring clean and cached branches.
  auto dirty_substages = RemoveCleanSubStages(all_substages, cached_imgs);

  // Search suitable packing under Vulkan limitations.
  //   Packing strategy is same as 'SearchSuitableFunctionPacking()'.
  Stages stages = SearchSuitableSubStagePacking(dirty_substages);
  return stages;
}
```

1.10. Vulkan Object Creation

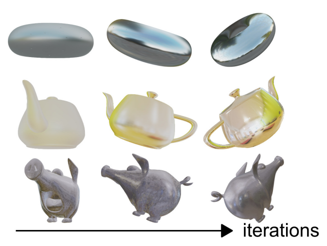
After stage packing, stages and substages are parsed to Vulkan objects. The input and output variables of substages are allocated as `VkImage` on GPU. We use only `VkImage` instead of `VkBuffer` because a graphics pipeline is more efficient than a compute pipeline and only a fragment shader can have multiple outputs as images. To simplify the implementation, non-image data such as vertex attributes are also treated as images. Stages for rasterization and fragment shaders are parsed into `VkRenderPass` objects, and substages are parsed into `VkPipeline` to be a subpass in a `VkRenderPass`. For a compute shader, one stage should have only one substage because there is no hierarchical structure. Therefore, the stage is parsed as one `VkPipeline`.

```
auto ParseStagesAsVulkanObjects(Stages stages,
                                std::map<Variable, VkImage> prev_vk_imgs,
                                std::map<Variable, Variable> upd_inp_map) {
  // Collect image usages for substage I/O
  std::map<Variable, VkImageUsage> usages = CollectVkImageUsage(stages);
  // Create images with usages
  std::map<Variable, VkImage> vk_imgs;
  for (auto [var, usage]: usages) {
    if (prev_vk_imgs.contains(var)) {
      // Skip image creation to keep previous image data
      vk_imgs[var] = prev_vk_imgs[var];
    } else if (upd_inp_map.contains(var)) {
      // Use the same image for input and updated variables.
      // Inputs will be overwritten by updated ones for each iteration.
      vk_imgs[var] = upd_inp_map(var);
    } else {
      // Create a new Vulkan image on GPU
      vk_imgs[var] =
          VkCreateImage(var.getVType(), var.getImgSize(), usage, ...);
    }
  }

  // Create pipelines, render passes, and lots of other Vulkan objects
  VkCommandBuffer vk_cmd_buf;
  std::vector<VkRenderPass> vk_renderpasses;
  std::vector<VkPipeline> vk_pipelines;
  ...
  for (auto stage: stages) {
    // Build a graphics or compute pipeline according to stage type.
    ShaderType stage_type = GetShaderType(stage);
    if (stage_type == FRAG) { // Including 'RASTER' too. Rasterization and
                              // fragment functions were packed into one
                              // stage.
      // Create a render pass from a stage for graphics pipelines.
      auto vk_renderpass = CreateRenderPass(stage);
      ...  // Many Vulkan setups
      VkCmdBeginRenderPass(vk_cmd_buf, vk_renderpass);

      // Subpass creation and recording
      for (auto substage: stage.substages) {
        // Create a pipeline for a substage as a subpass in the renderpass.
        auto vk_pipeline = CreateGraphicsPipeline(substage, vk_renderpass);
        vk_pipelines.push_back(vk_pipeline);
        ...  // Many Vulkan setups
        VkCmdBindPipeline(vk_cmd_buf, vk_pipeline);

        // Record drawing call
        if (substage.vtx_vars.empty()) {
          // Drawing for only fragment functions.
          //  No vertex buffer are bound, and dummy vertex shader is
          //  attached to rasterize a full-screen rectangle.
          VkCmdDraw(vk_cmd_buf);
        } else {
          // Drawing for rasterization and fragment functions.
          //  In order to rasterize one vertex attribute, a vertex buffer
          //  is bound, and a pass-through vertex shader is attached.
          VkCmdBindVertexBuffers(vk_cmd_buf);  // Bind vertex buffer
          VkCmdDraw(vk_cmd_buf);
        }
        VkCmdNextSubPass(vk_cmd_buf);
      }

      VkCmdEndRenderPass(vk_cmd_buf);
      vk_renderpasses.push_back(vk_renderpass);
    } else if (stage_type == COMP) {
      // Create a compute pipeline. It is slower than graphics pipelines.
      // 'substages.size() == 1' because of no subpass for compute shaders.
      auto vk_pipeline = CreateComputePipeline(stage.substages[0]);
      vk_pipelines.push_back(vk_pipeline);
      ...  // Many Vulkan setups
      VkCmdBindPipeline(vk_cmd_buf, vk_pipeline);
      // Record drawing call
      VkCmdDispatch(vk_cmd_buf);
    }
  }
  return vk_imgs, vk_cmd_buf, vk_renderpasses, vk_pipelines, ...;
}
```

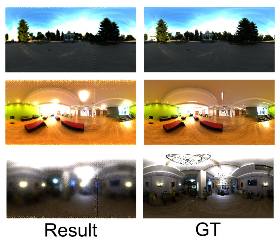
2. Additional Applications

2.1. Environment Map Optimization

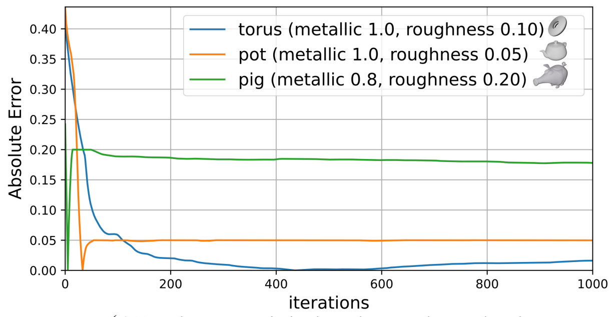
To demonstrate the versatility of our renderer, we jointly optimize an environment map and the physically-based shading (PBS) [MHM*13] roughness property. We use three meshes whose materials have different metallic and roughness properties and the environment maps provided by a public repository [Zaa21]. Fixed 20 views for 1,000 iterations are used to fit the pre-generated images rendered with GTs. We use Adam [KB14] optimizer in the optimization process. The environment map is initialized to a uniform gray color, and the minimum and maximum pixel intensities of the GT environment map are clamped to avoid noisy results. Fig. 1 shows that every combination of the mesh and the environment map converges to GT value. We observe some limitations through this experiment. First, we cannot fit optimized values to ground truth values such as low metallic properties, high roughness properties, or highly complicated geometries. It is impossible to optimize the environment map correctly with the matte materials due to the lack of visual cues in the rendered images. Second, the optimization contains artifacts that are caused by sampling patterns. There is room for improvement in the environment sampling method for optimization.



(a) Roughness changes during optimization



(b) Optimized environment map and GT



(c) Error between optimized roughness and ground truth

Figure 1: *Joint optimization of the PBS roughness property and the environment map: (a) changes in roughness through optimization, (b) optimized results and GTs of the environment maps, and (c) error between optimized roughness and GT for the meshes that are torus, pot, and pig [Cra21].*

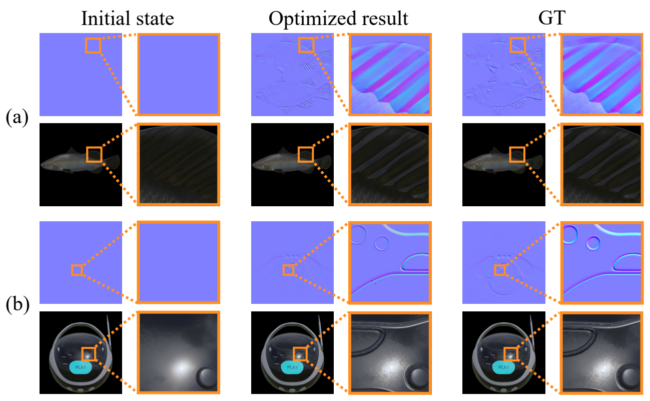
2.2. Normal Map Optimization

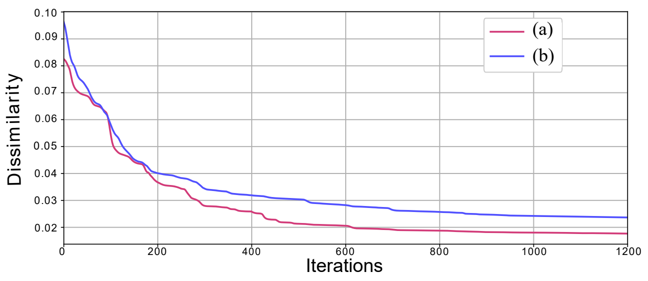
In this section, we demonstrate that Dressi can optimize a normal map to add fine details. We optimize the normal map of two geometries to fit the rendered image using a ground truth normal map. According to the norm constraint, we normalize the vector corresponding to each pixel every iteration. The optimization results of the fish and boombox [Khr21] models are shown in Fig. 2. The normal map is initialized in a uniform light blue color which represents the normal vector pointing directly to the viewer. Then, we rotate the camera pose in the render for every frame to cover the geometry surface as much as possible. In each frame, we calculate the RGB rendering loss L_{rbg} with Adam [KB14] optimizer. The RGB rendering loss is the ℓ_2-norm of the difference between the rendered image using the ground truth normal map $\mathbf{N}_{GT}$ and that with the optimized normal map $\mathbf{N}_\theta$ parameterized by $\theta \in \Theta$. In the upper part of Fig. 2, the optimized normal map shows a similar pattern with the ground truth. Considering the fin of the fish in Fig. 2 (a) and grooves of the boombox in Fig. 2 (b) as examples, the rendered images show the same detail with the ground truth, indicating the effectiveness of the normal map optimization.

To evaluate the dissimilarity between the optimized normal map $\mathbf{N}_\theta$ and the ground truth normal map $\mathbf{N}_{GT}$, we use the cosine dissimilarity value L_{dissim} as a metric. This is defined as:

$$L_{dissim}(\theta) = \sum_{p \in P(\mathbf{N}_{GT})} (1 - cos(\mathbf{N}_\theta(p), \mathbf{N}_{GT}(p)))/\|P(\mathbf{N}_{GT})\|. \quad (1)$$

where $P(\mathbf{N}_{GT})$ is a set of pixel positions in a non-flat region and p is a pixel position in the ground truth $\mathbf{N}_{GT}$.





(c) Dissimilarity between optimized normal map and GT

Figure 2: *Normal map optimization with fish and boombox models. The upper part shows the normal maps and rendered images of (a) fish and (b) boombox. The lower part (c) shows the numerical evaluation with cosine dissimilarity. The optimization starts from a uniform color image and finally generates a similar pattern with the ground truth after 1200 iterations. The cosine dissimilarity decreases with each iteration. This indicates that the optimized normal vector turns to be very close to the ground truth.*

Based on this metric, the bottom curve of Fig. 2 illustrates that the dissimilarity decreases in 1,200 iterations, and it finally converges to a small value. The dissimilarity cannot converge to zero due to two reasons. First, we fix the environment map as a uniform gray color during the optimization. The optimized normal map will overfit this lighting condition. Second, we rotate the camera pose uniformly. There is no guarantee that every tiny surface is optimized because of the sparse views. In spite of the limitations, the experimental results show that Dressi can optimize a pixel-wise normal map to enhance the fine details.

2.3. Human Face Modeling with 3D Morphable Model

To illustrate practical use of our approach in computer vision tasks, we apply Dressi to human face modeling using the 3D Morphable Model (3DMM) [EST*20]. In this experiment, we use a public selfie RGB-depth sequence [BLC*21] and our original sequence captured in the same manner. Their resolution is 480×640 and the intrinsic parameters are pre-calibrated. In those sequences, a subject performs head rotation in front of a fixed camera. Our method chooses four frames as input images by following [BLC*21]. Because a large difference in the rendering results is caused by view-direction, normal, and metallic-roughness components, it is good to use multiple views to estimate the view-independent values.

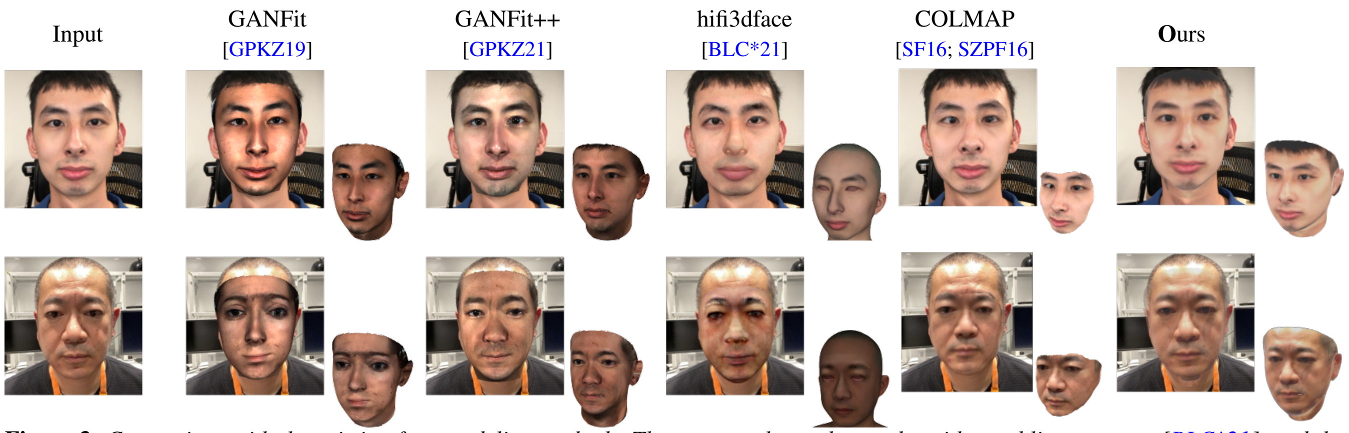


Figure 3: *Comparison with the existing face modeling methods. The top row shows the results with a public sequence [BLC*21], and the bottom row shows the results with our original sequence. GANFit [GPKZ19] and GANFit++ [GPKZ21] are based on generative adversarial network (GAN) and DR. They use a single RGB input and estimate shape, texture, and lighting. hifi3dface [BLC*21] with four view inputs minimizes a loss with 3DMM by DR and refines textures by GAN. COLMAP [SF16; SZPF16] is a traditional SfM+MVS pipeline. We input 100 uniformly sampled RGB images and use additional face region masks for COLMAP. Its output mesh is textured by MVS-texturing [WMG14]. Ours optimizes 3DMM by DR considering the normal and metallic-roughness textures with four views. Our results show a good similarity to subjects in the input images.*

Our optimization process initializes albedo, normal, and metallic-roughness textures with flat values. We initialize the camera poses of the four views and the scale of the 3DMM shape using the correspondences of detected 3D facial landmarks. Then, we start the optimization of the camera poses, scale, PCA coefficients of 3DMM as blendshape weights, and the textures. Our method uses SGD [SZ13] to optimize the textures, and Adam [KB14] is used for the other parameters. We use the fixed environment map as a uniform gray color and do not blur rendered images by HardSoftRas. We minimize the loss L as follows:

$$L = w_l L_l + w_d L_d + w_c L_c + w_r L_r. \tag{2}$$

In this equation, L_l is a 2D landmark loss with L1 norm, L_d denotes a pixel-wise L1 depth loss with truncation, L_c is a pixel-wise L2 RGB color loss, L_r represents a regularization term for the PCA coefficients and those textures, and w_l, w_d, w_c, and w_r are weights for the losses. To render color images for L_c, we use PBS with albedo, normal, and metallic-roughness textures. After the optimization, we apply simple image processing to the optimized textures to alleviate noises around the partially occluded areas for multiple views. Fig. 3 shows the comparison with the existing methods. Our method shows good geometry and texture reconstruction results with high visual similarity to the input image. However, it is difficult for our optimization-based method to decompose materials correctly because of the fixed lighting and lack of data-driven priors. For example, white highlights are often baked in albedo textures. Lighting estimation [GSY*17], learned features [PVZ15], and material basis [NLGK18; BLC*21] can improve material decomposition.

References

[BLC*21] BAO, LINCHAO, LIN, XIANGKAI, CHEN, YAJING, et al. "High-Fidelity 3D Digital Human Head Creation from RGB-D Selfies". *ACM Transactions on Graphics* (2021) 7, 8.

[Cra21] CRANE, KEENAN. *Keenan's 3D Model Repository*. 2021. URL: `https : / / www . cs . cmu . edu / ~kmcrane / Projects / ModelRepository/` 7.

[EST*20] EGGER, BERNHARD, SMITH, WILLIAM A. P., TEWARI, AYUSH, et al. "3D Morphable Face Models - Past, Present, and Future". *ACM Trans. Graph.* 39.5 (2020). ISSN: 0730-0301 7.

[GPKZ19] GECER, BARIS, PLOUMPIS, STYLIANOS, KOTSIA, IRENE, and ZAFEIRIOU, STEFANOS. "Ganfit: Generative adversarial network fitting for high fidelity 3d face reconstruction". *CVPR*. 2019 8.

[GPKZ21] GECER, BARIS, PLOUMPIS, STYLIANOS, KOTSIA, IRENE, and ZAFEIRIOU, STEFANOS. "Fast-GANFIT: Generative Adversarial Network for High Fidelity 3D Face Reconstruction". *IEEE Transactions on Pattern Analysis & Machine Intelligence* 01 (May 2021), 1–1. ISSN: 1939-3539. DOI: `10.1109/TPAMI.2021.3084524` 8.

[GSY*17] GARDNER, MARC-ANDRÉ, SUNKAVALLI, KALYAN, YUMER, ERSIN, et al. "Learning to Predict Indoor Illumination from a Single Image". *ACM Trans. Graph.* 36.6 (Nov. 2017). ISSN: 0730-0301 8.

[KB14] KINGMA, DIEDERIK P. and BA, JIMMY. "Adam: A Method for Stochastic Optimization". (2014) 6–8.

[Khr21] KHRONOS GROUP. *glTF Sample Models*. 2021. URL: `https://github.com/KhronosGroup/glTF-Sample-Models` 7.

[MHM*13] MCAULEY, STEPHEN, HILL, STEPHEN, MARTINEZ, ADAM, et al. "Physically based shading in theory and practice". *ACM SIGGRAPH Courses*. 2013, 1–8 6.

[NLGK18] NAM, GILJOO, LEE, JOO HO, GUTIERREZ, DIEGO, and KIM, MIN H. "Practical SVBRDF Acquisition of 3D Objects with Unstructured Flash Photography". *ACM Trans. Graph.* 37.6 (2018). ISSN: 0730-0301 8.

[PVZ15] PARKHI, OMKAR M, VEDALDI, ANDREA, and ZISSERMAN, ANDREW. "Deep face recognition". (2015) 8.

[SF16] SCHÖNBERGER, JOHANNES LUTZ and FRAHM, JAN-MICHAEL. "Structure-from-Motion Revisited". *CVPR*. 2016 8.

[SZ13] SHAMIR, OHAD and ZHANG, TONG. "Stochastic Gradient Descent for Non-smooth Optimization: Convergence Results and Optimal Averaging Schemes". *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28. Proceedings of Machine Learning Research 1. Atlanta, Georgia, USA: PMLR, 2013, 71–79 8.

[SZPF16] SCHÖNBERGER, JOHANNES LUTZ, ZHENG, ENLIANG, POLLEFEYS, MARC, and FRAHM, JAN-MICHAEL. "Pixelwise View Selection for Unstructured Multi-View Stereo". *European Conference on Computer Vision (ECCV)*. 2016 8.

[WMG14] WAECHTER, MICHAEL, MOEHRLE, NILS, and GOESELE, MICHAEL. "Let There Be Color! — Large-Scale Texturing of 3D Reconstructions". *Proceedings of the European Conference on Computer Vision*. Springer, 2014 8.

[Zaa21] ZAAL, GREG. *HDRIHaven*. 2021. URL: https://hdrihaven.com/ 6.