Realtime Deformation of Constrained Meshes Using GPU

Alexandre Kaspar and Bailin Deng

Abstract Constrained meshes play an important role in freeform architectural design, as they can represent panel layouts on freeform surfaces. It is challenging to perform realtime manipulation on such meshes, because all constraints need to be respected during the deformation while the shape quality needs to be maintained. This usually leads to nonlinear constrained optimization problems, which are challenging to solve in real time. In this paper, we present a GPU-based shape manipulation tool for constrained meshes, using the parallelizable algorithm proposed in [8]. We discuss the main challenges and solutions for the GPU implementation, and provide timing comparison against a CPU implementation of the algorithm. Our GPU implementation significantly outperforms the CPU version, allowing realtime handle-based deformation for large constrained meshes.

1 Introduction

With the advances in computer-aided design tools, complex freeform shapes are becoming more and more popular in architectural design nowadays. While digital models can be easily created using a computer, the construction of such shapes remains a challenge, due to limitation of fabrication technologies. To realize freeform architectural designs at a reasonable cost, the design surfaces usually need to be decomposed into panels of simple shapes that facilitate manufacturing. This process is called *rationalization*, which amounts to approximating the NURBS-based design surface using a set of panels subject to requirements such as approximation toler-

---

Alexandre Kaspar
Computer Graphics and Geometry Laboratory, École Polytechnique Fédérale de Lausanne, e-mail: alexandre.kaspar@a3.epfl.ch

Bailin Deng
Computer Graphics and Geometry Laboratory, École Polytechnique Fédérale de Lausanne, e-mail: bailin.deng@epfl.ch

Fig. 1: Panel layouts can be represented by polygonal meshes subject to geometric constraints. Left: Yas Viceroy Hotel in Abu Dhabi, designed by Asymptote Architecture (image courtesy of Asymptote Architecture). Right: a quad mesh representing the hotel facade, with the constraint that the vertices of each face lie on a common plane. This constrained mesh represents a layout of planar quadrilateral panels on the facade.

ance, panel types, aesthetics of panel layouts, etc. Rationalization usually involves nonlinear optimization with a large number of variables, and is therefore computationally expensive [11].

From a designer's point of view, it is important to explore different design shapes and their corresponding panel layouts. One possible way is to modify the NURBS design and perform rationalization for each new shape. Due to the heavy computational cost of rationalization, it is time-consuming to explore designs via this approach. An alternative approach is to directly manipulate the panel shapes and layouts, while respecting the shape requirements for panel types and maintaining the aesthetics of the overall shape. In this way the user only explores panel layouts that satisfy all the requirements, with intuitive feedback about what modifications are possible under the given requirements. Such *fabrication-aware* shape exploration methods for freeform architecture have been a popular research topic recently [25, 6, 24, 26, 19, 9].

Usually a panel layout can be represented by a polygonal mesh, with mesh faces representing the panels and mesh edges representing the panel boundaries. The shape requirements for panel layout induce geometric constraints for mesh elements. For example, a layout of planar panels corresponds to a polygonal mesh where the vertices of each face are required to be coplanar (see Figure 1). Therefore, manipulating the panel layout reduces to deforming the mesh while satisfying certain geometric constraints and maintaining the shape quality. This usually leads to a nonlinear constrained optimization problem for mesh vertex positions. Due to the difficulty of the optimization, it is a challenging task to perform realtime manipulation, especially for large meshes.

Bouaziz et al. [6] proposed a general framework for handle-based deformation of meshes subject to soft constraints, formulated as a nonlinear least squares problem. Utilizing projections of individual mesh elements onto their feasible configurations, they propose an iterative solver that alternates between global linear system solving and local mesh element projections. The projections are independent and can be executed in parallel, thus achieving significant speedup on multi-core processors. When run on a multi-core CPU, the method achieves interactive results for meshes with about 1K vertices, but still unable to handle large meshes. Recently, this method was extended in [8] to allow both hard and soft constraints. The proposed numerical solver consists of a series of simple subproblems similar to those in [6], enabling speedup from parallelism. In this paper, we present an implementation of the method in [8] on GPU using CUDA, which provides many more computational cores than CPU. By carefully optimizing for performance, our implementation allows realtime deformation of constrained meshes with up to 20K vertices and 20K constraints.

1.1 Related Work

Besides [6] and [8], other handle-based deformation methods for constrained meshes have been developed in recent years. Zhao et al. [26] extended the shape space exploration approach in [25], using curve handles for shape control. Vaxman [24] proposed a method to deform polyhedral meshes while keeping their faces planar, using affine transformations of mesh faces. The computation reduces to solving a linear system for mesh vertex positions, allowing realtime deformation. The method only works for polyhedral meshes (meshes with planar faces). Moreover, since only affine transformations are allowed, only a subset of the feasible deformations are considered, which limits the degree of freedom for shape control. Poranne et al. [19] provided an optimization approach to deform polyhedral meshes, not limited to affine transformations of faces. The deformation is computed through an alternating least-squares approach similar to [6]. However, only face planarity constraints are considered by the method. Deng et al. [9] proposed a framework to deform meshes under hard constraints, with a focus on computing local deformations. Their framework does not consider soft constraints. On the contrary, the deformation method in this paper considers general shape constraints for meshes, and allows both soft and hard constraints, providing more flexibility in shape manipulation.

Recently, computational design shape exploration tools have also been proposed for other types of architecture, such as reciprocal frame structures [21] and building layouts [1]. As these problems require other representations than polygonal meshes, they cannot be handled by our method.

1.2 Overview

The rest of the paper is organized as follows. Section 2 briefly presents the method in [8]. Section 3 gives an overview of the implementation of our system. Section 4 provides more details about the CUDA implementation. Finally, results are presented in Section 5, followed by a discussion about limitation and future work in Section 6. Section 7 concludes the paper.

2 Overview of the Method

In this section, we give a brief overview of the problem formulation in [8], as well as its numerical solution. Interested readers are referred to [8] for more details.

2.1 Problem Formulation

We consider polygonal meshes as a representation of panel layouts for freeform architectural surfaces. The mesh is deformed by changing its vertex positions while fixing its topology. During the deformation, the vertex positions are subject to certain soft constraints and/or hard constraints. To control the deformation, a user specifies target positions for some vertices using handles that are freely movable. When the handles are moved, the mesh vertex positions are updated such that:

- The new mesh satisfies the soft constraints as much as possible, and satisfies the hard constraints strictly;
- The handle vertices are close to their target positions;
- The non-handle vertices stay close to their original positions;
- The vertex deformation field is smooth across the mesh.

With a given topology, the shape of a mesh is determined by its vertex positions $\mathbf{p}_1, \ldots, \mathbf{p}_N \in \mathbb{R}^3$ where N is the number of vertices. A shape constraint involving m vertex positions $\mathbf{p}_{i_1}, \ldots, \mathbf{p}_{i_m}$ can be represented by the condition $(\mathbf{p}_{i_1}, \ldots, \mathbf{p}_{i_m}) \in \mathcal{C}$, where $\mathcal{C} \subset \mathbb{R}^{3m}$ is the feasible set. We assume that the constraint is translation-invariant, meaning that applying a common translation to all involved vertices does not change the status of constraint satisfaction (which is the case for most shape constraints relevant to freeform architecture). To facilitate numerical solution, we introduce auxiliary variables $\mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_m} \in \mathbb{R}^3$, and rewrite the constraint as

$$(\mathbf{y}_{i_1} \ldots \mathbf{y}_{i_m}) \in \mathcal{C}, \tag{1}$$

$$\mathbf{p}_j - \mathbf{mean}(\mathbf{p}_{i_1}, \ldots, \mathbf{p}_{i_m}) = \mathbf{y}_j, \quad \text{for } j = i_1, \ldots, i_m, \tag{2}$$

where $\mathbf{mean}(\mathbf{p}_{i_1}, \ldots, \mathbf{p}_{i_m}) = \dfrac{1}{m}(\mathbf{p}_{i_1} + \cdots + \mathbf{p}_{i_m})$ is the barycenter of $\mathbf{p}_{i_1}, \ldots, \mathbf{p}_{i_m}$. Note that (2) is a linear condition, and can be written in a matrix form $\mathbf{A}_{\mathcal{C}}\mathbf{p} = \mathbf{y}_{\mathcal{C}}$ where vector $\mathbf{p} \in \mathbb{R}^{3N}$ packs all vertex positions, vector $\mathbf{y}_{\mathcal{C}} \in \mathbb{R}^{3m}$ packs the auxiliary variables, and matrix $\mathbf{A}_{\mathcal{C}} \in \mathbb{R}^{3m \times 3N}$. For each soft constraint with feasible set $\mathcal{S}$, we introduce auxiliary variables $\mathbf{y}_{\mathcal{S}} \in \mathcal{S}$ to derive an equivalent condition $\mathbf{A}_{\mathcal{S}}\mathbf{p} = \mathbf{y}_{\mathcal{S}}$. Then the constraint violation can be measured with a function $F_{\mathcal{S}} = \|\mathbf{A}_{\mathcal{S}}\mathbf{p} - \mathbf{y}_{\mathcal{S}}\|_2^2$. Similarly for each hard constraint with feasible set $\mathcal{H}$, we introduce auxiliary variables $\mathbf{y}_{\mathcal{H}} \in \mathcal{H}$ to derive its equivalent condition $\mathbf{A}_{\mathcal{H}}\mathbf{p} = \mathbf{y}_{\mathcal{H}}$. Given N_s soft constraints and N_h hard constraints with feasible sets $\{\mathcal{S}_j \mid j = 1, \ldots, N_\mathrm{s}\}$ and $\{\mathcal{H}_k \mid k = 1, \ldots, N_\mathrm{h}\}$ respectively, the vertex positions $\mathbf{p}$ are computed by the following optimization

$$\begin{aligned} \min_{\mathbf{p},\mathbf{y}} \quad & w_{\mathrm{h}} F_{\mathrm{handle}} + w_{\mathrm{c}} F_{\mathrm{close}} + w_{\mathrm{f}} F_{\mathrm{fair}} + \sum_{j=1}^{N_{\mathrm{s}}} w_j^s F_{\mathcal{S}_j} + \sum_{j=1}^{N_{\mathrm{s}}} \sigma_{\mathcal{S}_j}(\mathbf{y}_{\mathcal{S}_j}) + \sum_{k=1}^{N_{\mathrm{h}}} \sigma_{\mathcal{H}_k}(\mathbf{y}_{\mathcal{H}_k}) \\ \text{s.t.} \quad & \mathbf{B}\mathbf{p} = \mathbf{y}_H. \end{aligned}$$

Here $\mathbf{y} = [\mathbf{y}_{\mathcal{S}_1}, \dots, \mathbf{y}_{\mathcal{S}_{N_{\mathrm{s}}}}, \mathbf{y}_{\mathcal{H}_1}, \dots, \mathbf{y}_{\mathcal{H}_{N_{\mathrm{h}}}}]$ packs all auxiliary variables for soft constraints and hard constraints, $F_{\mathcal{S}_j}$ is the soft constraint violation function introduced above, and side condition $\mathbf{B}\mathbf{p} = \mathbf{y}_H$ collects all linear relations from the equivalent conditions of hard constraints, with $\mathbf{B} = [\mathbf{A}_{\mathcal{H}_1}^T, \dots, \mathbf{A}_{\mathcal{H}_{N_{\mathrm{h}}}}^T]^T$ and $\mathbf{y}_H = [\mathbf{y}_{\mathcal{H}_1}^T, \dots, \mathbf{y}_{\mathcal{H}_{N_{\mathrm{h}}}}^T]^T$. Functions F_{handle}, F_{close}, F_{fair} measure the distance from handle vertices to their target positions, the distance from non-handle vertices to their original positions, and the smoothness of the vertex deformation field based on its Laplacian, respectively,

$$F_{\mathrm{handle}} = \sum_{i \in \Gamma} \|\mathbf{p}_i - \mathbf{t}_i\|_2^2, \quad F_{\mathrm{close}} = \sum_{j \notin \Gamma} \|\mathbf{p}_j - \mathbf{p}_j^0\|_2^2, \quad F_{\mathrm{fair}} = \|\mathbf{L}(\mathbf{p} - \mathbf{p}^0)\|^2,$$

where Γ is the index set for handle vertices, $\mathbf{t}_i$ is the target position for vertex i, $\mathbf{p}_j^0$ is the original position for vertex j, $\mathbf{p}^0$ packs the original positions for all vertices, and $\mathbf{L}$ is the Laplacian matrix. The *indicator function* $\sigma_{\mathcal{S}_j}(\mathbf{y}_{\mathcal{S}_j})$ makes sure $\mathbf{y}_{\mathcal{S}_j} \in \mathcal{S}_j$ in the solution, with

$$\sigma_{\mathcal{S}_j}(\mathbf{y}_{\mathcal{S}_j}) = \begin{cases} 0, & \text{if } \mathbf{y}_{\mathcal{S}_j} \in \mathcal{S}_j, \\ +\infty, & \text{otherwise.} \end{cases}$$

Indicator function $\sigma_{\mathcal{H}_k}(\mathbf{y}_{\mathcal{H}_k})$ is defined in the same way. w_{h}, w_{c}, w_{f} and w_j^s are positive weights trading off different terms. The optimization problem can be written in matrix form as

$$\begin{aligned} & \min_{\mathbf{p},\mathbf{y}} \|\mathbf{D}\mathbf{p} - \mathbf{r}\|_2^2 + w_{\mathrm{f}} \|\mathbf{L}(\mathbf{p} - \mathbf{p}^0)\|_2^2 + \sum_{j=1}^{N_{\mathrm{s}}} w_j^s \|\mathbf{A}_{\mathcal{S}_j}\mathbf{p} - \mathbf{y}_{\mathcal{S}_j}\|_2^2 + \sum_{j=1}^{N_{\mathrm{s}}} \sigma_{\mathcal{S}_j}(\mathbf{y}_{\mathcal{S}_j}) + \sum_{k=1}^{N_{\mathrm{h}}} \sigma_{\mathcal{H}_k}(\mathbf{y}_{\mathcal{H}_k}) \\ & \text{s.t. } \mathbf{B}\mathbf{p} = \mathbf{y}_H, \end{aligned} \tag{3}$$

where

$$\mathbf{D} = \begin{bmatrix} d_1 \mathbf{I}_3 & & \\ & \ddots & \\ & & d_N \mathbf{I}_3 \end{bmatrix}, \qquad \mathbf{r} = \begin{bmatrix} \mathbf{r}_1 \\ \vdots \\ \mathbf{r}_N \end{bmatrix},$$

with $\mathbf{I}_3$ being the 3×3 identity matrix, and

$$d_i = \begin{cases} \sqrt{w_{\mathrm{h}}} & \text{if } i \in \Gamma \\ \sqrt{w_{\mathrm{c}}} & \text{otherwise} \end{cases}, \qquad \mathbf{r}_i = \begin{cases} d_i \mathbf{t}_i & \text{if } i \in \Gamma \\ d_i \mathbf{p}_i^0 & \text{otherwise} \end{cases}, \qquad \text{for } i = 1, \dots, N.$$

2.2 Numerical Solution

2.2.1 Alternating Minimization

Without hard constraints, problem (3) reduces to minimizing quadratic terms with indicator functions. It is solved by alternating between two steps until convergence

1. *Projection*: fix $\mathbf{p}$, minimize over $\mathbf{y}$;
2. *Linear solve*: fix $\mathbf{y}$, minimize over $\mathbf{p}$.

The minimization in Step 2 simply amounts to solving a symmetric positive definite (SPD) sparse linear system, hence the name. For Step 1, the problem is separable for auxiliary variables from different constraints, and is solved in parallel. Specifically, we solve a set of independent subproblems, each of which is associated with one constraint and has the following form

$$\min_{\mathbf{y}_{\mathcal{C}}} \|\mathbf{y}_{\mathcal{C}} - \mathbf{x}\|_2^2 + \sigma_{\mathcal{C}}(\mathbf{y}_{\mathcal{C}}),$$

where $\mathcal{C}$ is the feasible set and $\mathbf{y}_{\mathcal{C}}$ are the auxiliary variables for the constraint. The solution is the closest *projection* from $\mathbf{x}$ onto $\mathcal{C}$, which we call the *proximal operator* of $\mathcal{C}$ for input data $\mathbf{x}$. For many constraints, we can derive the close-form representation of the proximal operator. For example (see [6] for details):

- **Coplanarity**. This constraint requires $n > 3$ vertices to lie on a common plane. It can be used to model planar panels, for example, by requiring the vertices of each mesh face to be coplanar (see Figure 1). The proximal operator finds n coplanar points $\mathbf{y}_1, \ldots, \mathbf{y}_n \in \mathbb{R}^3$ which are closest to the input data $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^3$. The solution is
$$\mathbf{y}_i = \mathbf{x}_i - \mathbf{n}[\mathbf{n} \cdot (\mathbf{x}_i - \overline{\mathbf{x}})], \qquad i = 1, \ldots, n,$$
where $\overline{\mathbf{x}} = \mathbf{mean}(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, and $\mathbf{n}$ is the left singular vector of matrix $[\mathbf{x}_1, \ldots, \mathbf{x}_n]$ for the smallest singular value.
- **Regular Polygon**. This constraint requires a face with $n \geq 3$ vertices to be a regular n-gon. It can be used to induce shape regularity of mesh elements (see Figure 2). The proximal operator finds a regular n-gon closest to a polygon with vertices $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^3$. This can be done by computing the translation, rotation and scaling of a predefined regular n-gon to fit the target polygon, using the algorithm in [23].

2.2.2 Augmented Lagrangian Method

When dealing with hard constraints, extra care has to be taken to ensure that the linear side constraints in problem (3) is satisfied. This is done using the *augmented Lagrangian method* (ALM) [5], which searches for a saddle point of the following *augmented Lagrangian function*

$$\mathcal{L}(\mathbf{p}, \mathbf{y}, \boldsymbol{\lambda}; \mu) = F(\mathbf{p}, \mathbf{y}) + \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{p}, \mathbf{y}) + \mu \|\mathbf{h}(\mathbf{p}, \mathbf{y})\|_2^2, \tag{4}$$

where $F(\mathbf{p}, \mathbf{y})$ is the target function in (3), and $\mathbf{h}(\mathbf{p}, \mathbf{y}) = \mathbf{B}\mathbf{p} - \mathbf{y}_H$ is the residual of side constraints in (3), $\boldsymbol{\lambda}$ is a vector of dual variables, and $\mu > 0$ is a penalty parameter. The solver iteratively updates $\mathbf{p}, \mathbf{y}, \boldsymbol{\lambda}$ and μ until convergence. In each iteration, new values $\hat{\mathbf{p}}, \hat{\mathbf{y}}, \hat{\boldsymbol{\lambda}}, \hat{\mu}$ are computed from current values $\overline{\mathbf{p}}, \overline{\mathbf{y}}, \overline{\boldsymbol{\lambda}}, \overline{\mu}$ using the following steps

1. *Primal update*: $(\hat{\mathbf{p}}, \hat{\mathbf{y}}) = \operatorname{argmin}_{\mathbf{p}, \mathbf{y}} \mathcal{L}(\mathbf{p}, \mathbf{y}, \overline{\boldsymbol{\lambda}}; \overline{\mu})$;
2. *Dual update*: $\hat{\boldsymbol{\lambda}} = \overline{\boldsymbol{\lambda}} + \overline{\mu}\, \mathbf{h}(\hat{\mathbf{p}}, \hat{\mathbf{y}})$;
3. *Penalty update*: choose $\hat{\mu} \geq \overline{\mu}$.

The problem in Step 1 has a similar structure as the one from Section 2.2.1, and is solved in the same way. Specifically, it alternates between two steps

1. Projection step with proximal operator evaluations

$$\min_{\mathbf{y}_{\mathcal{S}_j}} \|\mathbf{y}_{\mathcal{S}_j} - \mathbf{A}_{\mathcal{S}_j} \mathbf{p}\|_2^2 + \sigma_{\mathcal{S}_j}(\mathbf{y}_{\mathcal{S}_j}), \qquad j = 1, \ldots, N_{\mathrm{s}},$$

$$\min_{\mathbf{y}_{\mathcal{H}_k}} \|\mathbf{y}_{\mathcal{H}_k} - (\mathbf{A}_{\mathcal{H}_k} \mathbf{p} + \frac{\boldsymbol{\lambda}_{\mathcal{H}_k}}{2\mu})\|_2^2 + \sigma_{\mathcal{H}_k}(\mathbf{y}_{\mathcal{H}_k}), \qquad k = 1, \ldots, N_{\mathrm{h}},$$

   where $\boldsymbol{\lambda}_{\mathcal{H}_k}$ collects the components of $\boldsymbol{\lambda}$ in the same positions as $\mathbf{y}_{\mathcal{H}_k}$ in $\mathbf{y}_H$.
2. Solving a sparse SPD system for $\mathbf{p}$

$$\begin{aligned}&\left(\mathbf{D}^T\mathbf{D} + w_{\mathrm{f}}\mathbf{L}^T\mathbf{L} + \mu\mathbf{B}^T\mathbf{B} + \sum_{j=1}^{N_{\mathrm{s}}} w_j^s \mathbf{A}_{\mathcal{S}_j}^T \mathbf{A}_{\mathcal{S}_j}\right)\mathbf{p} \\ &= \mathbf{D}^T\mathbf{r} + w_{\mathrm{f}}\mathbf{L}^T\mathbf{L}\mathbf{p}^0 + \mu\mathbf{B}^T\left(\mathbf{y}_H - \frac{\boldsymbol{\lambda}}{2\mu}\right) + \sum_{j=1}^{N_{\mathrm{s}}} w_j^s \mathbf{A}_{\mathcal{S}_j}^T \mathbf{y}_{\mathcal{S}_j}.\end{aligned} \tag{5}$$

The primal update step is the most time-consuming part of the solver. We will not go into the details of Steps 2 and 3, but refer the readers to [8] instead. Note that for a given problem, the linear system matrix in problem (5) only changes according to penalty parameter μ. The penalty update scheme in [8] only generates a predefined set of values for μ, so we can precompute all linear system matrices that appear in problem (5).

3 General Implementation Strategies

We developed an interactive handle-based shape manipulation system for constrained meshes, based on the algorithms presented in the previous section. For an initial mesh, the user selects a set of handle vertices, and specify their target positions (which we call *handle positions*) by dragging 3D manipulators. Whenever the

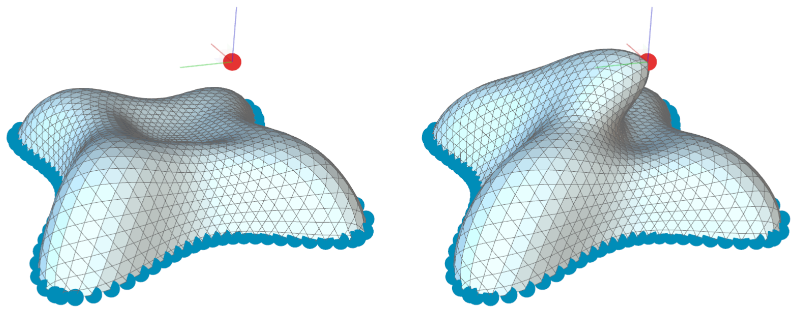


Fig. 2: Handle-based deformation of a constrained mesh subject to the soft constraint that each faces is a regular polygon. Left: the initial mesh with the handles (shown in red and blue) attached to the boundary vertices and four vertices in the middle. The handles for the middle positions are moved to new target positions (shown in red). Right: the mesh deforms according to the handle positions, while satisfying the soft constraints.

manipulators are moved, the system deforms the mesh according to the new handle positions, providing immediate feedback to the user (See Figure 2 for an example).

Figure 3 shows the architecture of our system. Here we distinguish between the work of the threads from the user side (user interface, mouse and keyboard interaction, mesh display, etc.) which we gather as the *user module*, and the work done within a single thread dedicated to a GPU-based ALM solver which we call the *optimization module*. The latter loops over three main logical steps:

1. *Input phase*: transfer current handle positions to GPU;
2. *Optimization phase*: iterate the ALM steps on GPU, until some output conditions are satisfied;
3. *Output phase*: read back updated vertex positions from GPU.

To run the ALM solver on GPU, we store on the GPU memory all the optimization variables, as well as other auxiliary data (such as matrices $\mathbf{A}_{\mathcal{S}_j}$, $\mathbf{B}$, $\mathbf{D}$ and vector $\mathbf{r}$ in formulation (3), and the linear system matrices in problem (5)). Many of these data remain constant during optimization, and only need to be initialized once at the beginning. Thus in the input phase, we only need to transfer the latest handle positions to the GPU to update the problem specification.

As an iterative solver, the optimization phase requires initial values of the variables. To initialize the current optimization phase, we always use the resulting variable values from the previous optimization phase. The motivation is that when a user drags the handles continuously, the handle positions used in two consecutive optimization phases are close to each other. Thus their solutions will be close to each other as well, making the solution from the previous phase a good guess for the current solution.

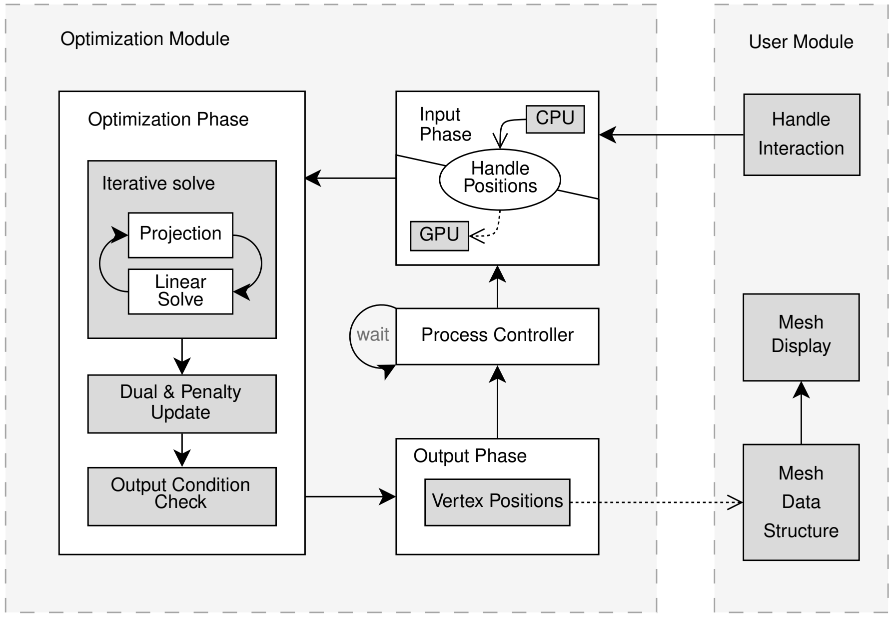


Fig. 3: The architecture of our GPU-based implementation.

Depending on the data, the optimization phase might take a large number of iterations to fully converge. To keep the process interactive, we allow switching from optimization phase to output phase even if it is not fully convergent yet. When the handles are dragged, they are likely to be moving at the same as the ALM solver is running. Rather than solving the current problem to a very high accuracy, it is more important to output the current result and start a new optimization phase with the new handle positions, so that the mesh shape follows the handle positions smoothly and shows how the shape reacts to handle position changes. Even if the output mesh shape is not the exact solution, it is still a good approximation because the solver usually converges quickly to an approximate solution [7]. Therefore, we switch from optimization phase to output phase, if one of the following conditions is satisfied

1. The optimization phase fully converges;
2. The number of iterations within the optimization phase exceeds a limit $M_{\max}$.

The output phase is responsible for reading back new vertex positions in order to update the mesh data structure in host memory, which is then used to update the mesh display. Both operations (vertex readback and mesh display update) involve data transfer between CPU and GPU. To avoid unnecessary transfer while keeping the process interactive, we only read back vertex positions if the elapsed time (in milliseconds) from the last readback is larger than a threshold ε. With such a strategy, the maximum frame rate for mesh display is $1000/\varepsilon$ FPS.

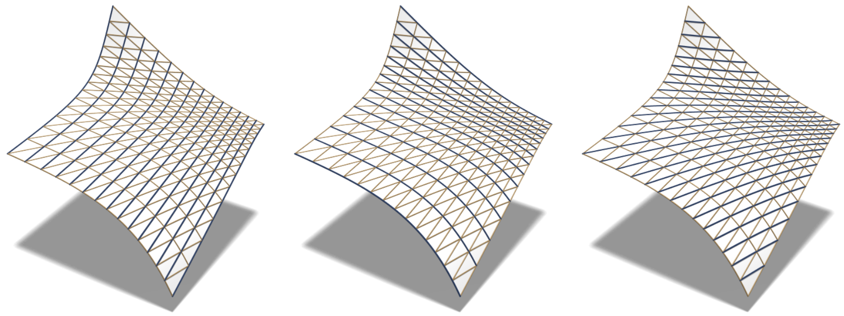


Fig. 4: For a regular triangle mesh (i.e., each interior vertex has valence six, and each boundary vertex has valence no larger than six), there exist three families of edge polylines (shown in blue). Being a planar web requires each polyline to be planar, namely all vertices on the polyline lie in a common plane.

After the output phase, depending on the availability of new handle positions and the convergence of the optimization phase, we are in one of the following cases

- If there are new handle positions, transfer them to GPU and start a new optimization phase;
- Otherwise, if the previous optimization phase was not fully convergent, resume the optimization;
- Otherwise, wait for new handle positions.

4 CUDA Implementation Details

Our GPU implementation was done with CUDA. We targeted NVIDIA GeForce GTX 580 [18], which runs under the Fermi architecture [17, 12]. It has 16 streaming multiprocessors providing a total of 512 cores. Each of them has 64kB of memory available between the L1 cache and the shared memory. The rest of this section will present the challenges and specific implementation details.

4.1 Kernels

We implemented custom kernels for two critical operations: updating the handle data and evaluating the proximal operators.

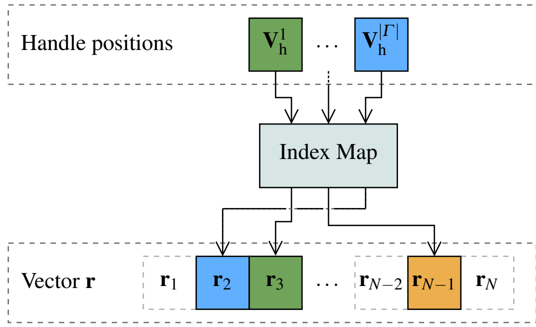


Fig. 5 The update of the handle data on GPU is done using a kernel that fills vector $\mathbf{r}$ according to the handle position vector (on GPU) and a precomputed index map.

4.1.1 Handle Update

When starting an optimization phase with new handle positions, we need to update the GPU memory storage of vector $\mathbf{r}$ in formulation (3). With the number of handle vertices being usually much smaller than the number of vertices, we first transfer the handle positions onto the GPU memory as a contiguous vector $\mathbf{V}_\mathrm{h} \in \mathbb{R}^{3|\Gamma|}$. Then a custom kernel updates the entries of $\mathbf{r}$ according to $\mathbf{V}_\mathrm{h}$, using a precomputed index map (see Figure 5). Note that the index map remains unchanged during optimization, since neither the choice of handle vertices nor the mesh topology is allowed to change.

Another strategy would be to transfer only the handle positions that are being changed by the user. This requires a *dynamic* index map for writing to vector $\mathbf{r}$, as well as checking which handles are being moved. To simplify implementation, we did not use such strategy.

4.1.2 Proximal Operator Evaluation

As we saw in Section 2, proximal operators are responsible for updating auxiliary variables. Each type of constraint corresponds to one proximal operator, which involves a predefined set of operations. For different constraints of the same type, their proximal operator evaluation is independent since the involved auxiliary variables do not overlap. Such characteristics make it suitable to evaluate proximal operators using custom CUDA kernels. Specifically, we implement one kernel for each type of constraint, within which each thread handles one constraint.

For high performance, we need to ensure coalesced memory access. Thus we store the auxiliary variables $\mathbf{y}$ in formulation (3) with a contiguous array in global memory, where the components corresponding to the same kernel reside in a contiguous region. The input data for proximal operators are of the same dimension as $\mathbf{y}$, and we store them with an array in global memory using the same layout as $\mathbf{y}$ (see Figure 6 for example).

Another performance consideration is the grid and block sizes. We follow [22] which suggests a number of threads per block:

1. Dividing the maximum number of threads per SM, i.e. 1536 for Fermi;
2. At least 32 threads per block, i.e. the warp size;
3. At most 8 blocks per SM, so as to maximize occupancy (and thus at least $1536/8 = 192$ threads per block).

Since we do not know the relation between different types of kernels, we chose to simply saturate them by using a block size of 512 threads, which proved to be sufficient for our need according to experiments.

Coplanarity Constraint

Because of specific features and limitations of GPU, additional care needs to be taken when implementing some proximal operators. Here we use the vertex coplanarity constraint as an example to show the challenges and our solutions. Coplanarity constraint is one of the most important shape constraints in freeform architecture. It can be used to model planar panels [13] (Figure 1), as well as *planar webs* which consist of curve elements of planar shapes [10] (Figure 4). For input data $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^3$, a key step of the proximal operator is a singular value decomposition (SVD), to extract the left singular vector of matrix $\mathbf{M} = [\mathbf{x}_1, \ldots, \mathbf{x}_n] \in \mathbb{R}^{3\times n}$ for the smallest singular value (see Section 2.2.1).

Due to the memory layout requirement mentioned before, the global memory storage of $\mathbf{x}_1, \ldots, \mathbf{x}_n$ is already a column-major representation for matrix $\mathbf{M}$. Thus a naïve approach is to implement an SVD solver that operates directly on the global memory storage of $\mathbf{M}$. However, this might lead to excessive access to global memory, lowering the performance significantly [16].

To reduce global memory access, we implemented the kernel as follows. First note that the target singular vector is the same as the right singular vector of 3×3 matrix $\mathbf{M}\mathbf{M}^T = \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^T$ for the smallest singular value. Thus we create matrix $\mathbf{M}\mathbf{M}^T$ on local memory, by reading each $\mathbf{x}_i$ from global memory and summing up $\mathbf{x}_i \mathbf{x}_i^T$. Afterwards, we perform SVD on matrix $\mathbf{M}\mathbf{M}^T$. In this way, each global memory element of $\mathbf{M}$ needs to be accessed only once for computing the singular vector. Moreover, this approach only performs SVD on a 3×3 matrix. For coplanarity constraints involving a large number of vertices, this significantly reduces the matrix storage on local memory, compared to the original matrix $\mathbf{M}$. Such compact storage helps to reduce register spilling and L1 cache miss, which improves the performance of the kernel. Furthermore, with this approach we are able to deal with coplanary constraints with different number of vertices using a single kernel, by precomputing an array that stores for each coplanarity constraint the following information: 1) the number of vertices; and 2) the address of input data. Using a single kernel helps to increase parallelism for the implementation, resulting in improved throughput of the system. Figure 6 provides a schematic diagram for the kernel of coplanarity constraints.

For 3×3 SVD, we implemented a simple SVD solver based on [20]. There exists a branch-free 3×3 SVD solver [15] that might provide higher performance, but our simple implementation turned out to be sufficient.

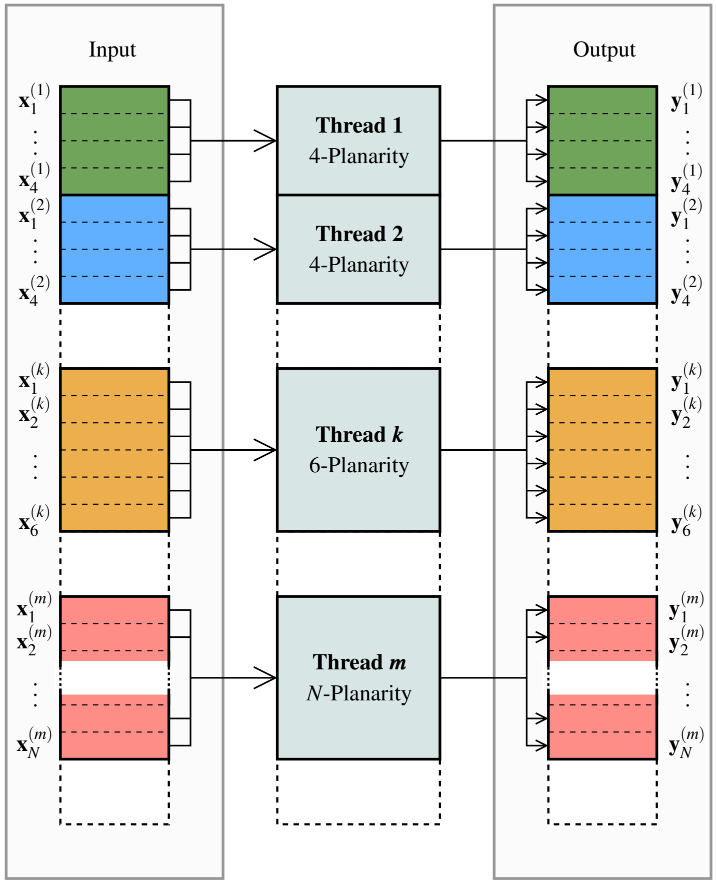


Fig. 6: Schematic diagram for the proximal operator kernel of coplanarity constraints. Input data $\mathbf{x}$ and output data $\mathbf{y}$ are stored in two contiguous arrays respectively. Within each array, data associated with a thread reside in a contiguous region. Our implementation is able to handle coplanarity constraints for different number of vertices within a single kernel. Here N-planarity refers to a coplanarity constraint for N vertices.

4.2 Sparse Linear Algebra

In general, all matrices in formulation (3) are sparse, while the vectors are all dense. Therefore, the solver requires many sparse matrix vector multiplications (SpMV). For these operations we used the Cusp library [4] which provides an easy C++ interface for sparse linear algebra with CUDA. Among the sparse matrix formats provided by Cusp, we chose the hybrid format (ELL + COO) as it provides faster linear operations for general unstructured sparse matrices [3].

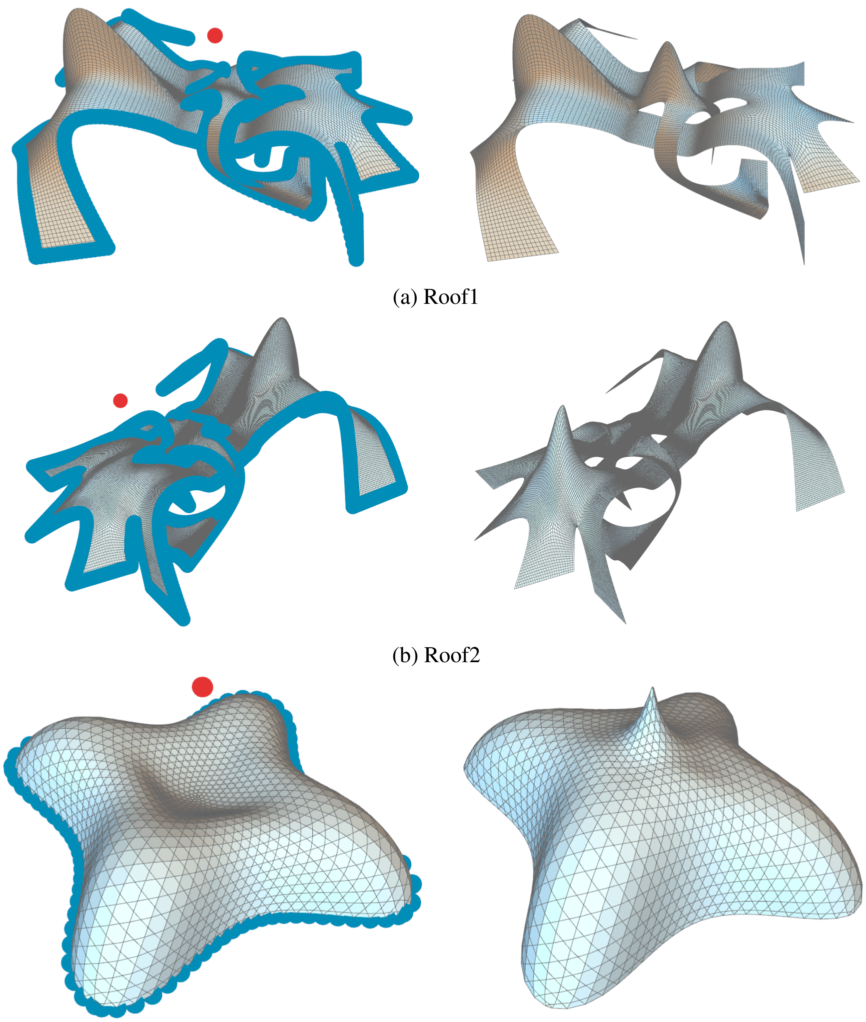
Since we are targeting large meshes, we solve the sparse linear system (5) using a conjugate gradient (CG) solver provided by Cusp. To warm-start the solver, we always use the previous CG solution as initial value for the current CG solving. Typically the right-hand side of system (5) changes gradually within the ALM solver, thus two consecutive solutions of problem (5) do not deviate significantly from each other, making this warm-starting strategy a reasonable choice. Alternatively, direct solvers based on Cholesky factorization can be more efficient. On the other hand, they often require more memory storage, because the sparsity of the linear system matrix is not preserved by its Cholesky factors. This could be an issue for GPU, since typically the amount of GPU memory is smaller than the host memory. Thus in our implementation we opted for a simple CG solver.

5 Results

In this section, we provide some performance results of our GPU-based constrained mesh deformation method, and compare them against the CPU version. The CPU version follows the same optimization workflow as described in Section 3, except that all the data reside in the host memory so there is no need to transfer handle positions in input phase and read back vertex positions in output phase. For both CPU and GPU versions, the framerate was limited to 30 FPS (i.e., the minimum elapsed time between two vertex readback operations is 33.3 ms), and the maximum number of iterations in optimization phase was set to $M_{\max} = 50$.

Both CPU and GPU versions were implemented for double-precision floating point data. We used two CPU implementations with different solvers for system (5): one uses CG, and the other uses a direct solver based on Cholesky factorization. Both CPU implementations reduce system (5) into three smaller systems for the x, y, z coordinates of the vertices respectively, with the same system matrix [6]. This allows the three coordinates of each vertex to be solved in parallel. The CPU version utilized OpenMP for the parallelization of proximal operator evaluation and linear system solving, and used the Eigen library [14] for linear algebra operations. For the CG solver on both CPU and GPU, we set the maximum number of iterations to 100, and the tolerance for the 2-norm ratio between the residual and right-handle side to 1×10^{-6}. The CPU and GPU implementations were run on a PC with an NVIDIA GeForce GTX 580 and an Intel Core i7 870 with four cores.

For comparison, each implementation was run with the same set of meshes and constraints. Since the optimization phase spends most of the running time on proximal operator evaluation and linear system solving, we focused the performance comparison on these two steps. Thus we only used soft constraints in our experiments, so that the optimization phase alternated between proximal operator evaluation and linear system solving. Figures 7 and 8 show the meshes used in our experiments, with the configuration of meshes and their constraints listed in Table 1. Here the initial mesh in Roof2 is a subdivided version of the initial mesh in Roof1, while Lilium1 and Lilium2 have the same initial mesh shape under different con-



(a) Roof1

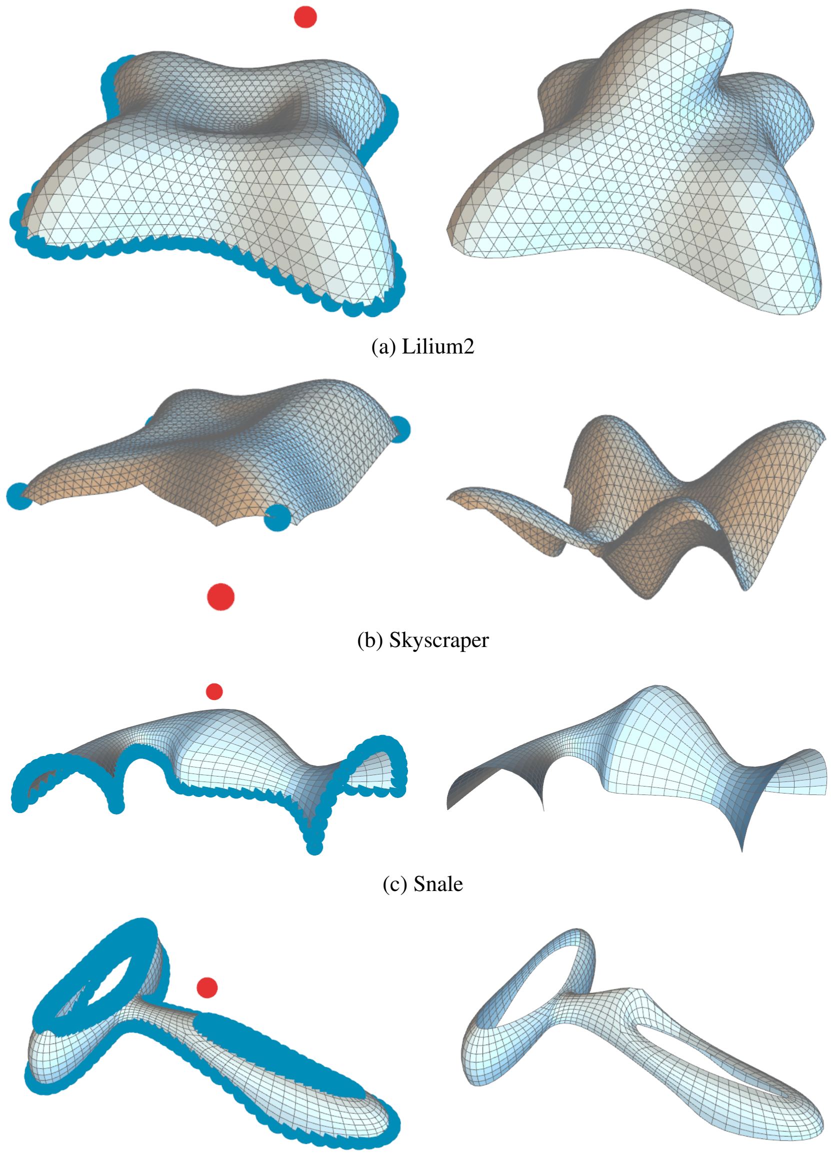
(b) Roof2

(c) Lilium1

Fig. 7: The first sets of models with their configuration and output illustration.

straints. The coplanarity constraints (for planar faces and planar web) are applied to a face or a polyline only if it has more than three vertices, while the constraints of regular polygons are applied to all faces of a mesh. For each mesh, some boundary vertices and interior vertices were chosen as handle vertices, with their handle positions shown in blue and red, respectively. In each experiment, the red handles were moved to trigger mesh deformation.

Table 2 shows the average elapsed time between two entries to the output phase, which we refer to as *average frame time*. A system with average frame time of α



(a) Lilium2

(b) Skyscraper

(c) Snale

(d) Yas

Fig. 8: The last sets of models with their configuration and output illustration.

milliseconds can achieve an average framerate up to $1000/\alpha$ FPS if the framerate is not limited. Thus smaller average frame time indicates more interactive result. We can see that even for a mesh with 80K vertices and 79K constraints, our GPU implementation achieves a framerate of 9 FPS, while the framerates for CPU imple-

Table 1: Configurations for meshes shown in Figures 7 and 8.

Reference Label	Vertices	Faces	Constraint Type	Handles
`Roof1`	20464	19712	Planar Faces	1505
`Roof2`	80352	78848	Planar Faces	3012
`Lilium1`	3504	3505	Regular Polygon Faces	100
`Lilium2`	3504	3505	Planar Faces	100
`Skyscraper`	1517	2884	Planar Web	5
`Snale`	1092	1020	Planar Faces	143
`Yas`	1085	976	Planar Faces	221

mentations are much lower than 1 FPS. For a smaller model with about 1K vertices and 1K constraints, our GPU implementation can potentially achieve a framerate of over 300 FPS, well beyond the specified upper limit. The comparison on average frame time shows that our GPU implementation gained significant speedups with respect to the CPU implementations.

The accompanying video shows the user interaction for Roof2. We can see that due to the large number of vertices and constraints, the CPU implementations failed to respond quickly to handle position changes. On the other hand, the GPU implementation remains interactive, leading to more intuitive shape manipulation.

Finally, Table 3 gives the timing ratio between input phase (*Input*), proximal operator evaluation (*Projection*), linear system solving (*CG*), and output phase (*Output*) for a typical interaction session on GPU. We can see that linear system solving spent the largest portion of time.

Table 2: Average frame time for different implementations.

Mesh	Average Frame Time [ms] CPU CG	CPU Cholesky	GPU CG
`Roof1`	2159.43	791.03	29.32
`Roof2`	14965.90	3842.25	107.20
`Lilium1`	638.45	132.84	17.82
`Lilium2`	210.34	43.99	14.01
`Skyscraper`	119.77	279.12	3.92
`Snale`	115.29	63.78	23.89
`Yas`	94.64	52.45	3.04

Table 3: Ratio of running time in each part of the optimization phase on GPU.

Mesh	% of the time spent in GPU optimization phase			
	Input	Projections	CG	Output
`Roof1`	0.34	4.27	88.29	7.10
`Roof2`	0.00	3.32	86.81	9.87
`Lilium1`	0.00	0.21	98.01	1.78
`Lilium2`	0.04	0.16	97.22	2.59
`Skyscraper`	0.00	0.85	98.43	0.72
`Snale`	0.01	0.04	99.89	0.07
`Yas`	0.28	0.74	98.71	0.28

6 Limitation and Future Work

In our system, the linear system solving is the bottleneck of performance. This is due to the well-known fact that SpMV involves irregular data access and thus achieves lower performance compared to dense operations on GPU. This motivates us to explore more advanced GPU SpMV techniques such as [2] to further optimize the performance. Another option is to adapt Cholesky-based direct solvers to GPU, as direct solvers outperformed CG for CPU implementations in many of our experiments.

A more ambitious improvement would be a hybrid GPU/CPU optimization. Currently the CPU is only used for managing the GPU, and it is mostly idle during the optimization. Thus we plan to investigate workload distribution between CPU and GPU to gain higher performance.

Our implementation requires frequent readback of vertex positions from GPU in order to update the display, which incurs some performance loss. One of our future plans is to directly update mesh display on GPU using vertex buffer object, thus totally avoiding data transfer between CPU and GPU in the output phase.

Finally, our system runs on CUDA-enabled GPUs only. We intend to develop an OpenCL-based system to make the algorithm available for a wider range of hardwares and platforms, and to compare the performance between different GPUs.

7 Conclusion

In this paper, we present an efficient handle-based constrained mesh manipulation system implemented on GPU. The mesh manipulation is formulated as a constrained optimization problem, which is decomposed into simple subproblems that can be solved in parallel. Utilizing the computational power of GPU, we achieve significant speedup of constrained mesh deformation compared to CPU implementation, as shown by our experiments on meshes with different sizes and constraints. On the other hand, linear system solving becomes the performance bottleneck, which provides an interesting avenue for future research.

Acknowledgements The authors thank Asymptote Architecture for providing the figure of Yas Viceroy Hotel. The mesh models are provided by Asymptote Architecture and Waagner Biro (Yas), Zaha Hadid Architects and Amir Vaxman (Lilium1 and Lilium2), and Mario Deuss (Roof1, Roof2 and Snale). This work has been supported by Swiss National Science Foundation (SNSF) grant 200021_137626.

References

1. Bao, F., Yan, D.M., Mitra, N.J., Wonka, P.: Generating and exploring good building layouts. ACM Trans. Graph. **32**(4), 122:1–122:10 (2013)
2. Baskaran, M.M., Bordawekar, R.: Optimizing sparse matrix-vector multiplication on GPUs. Tech. Rep. RC24704, IBM (2008)
3. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pp. 18:1–18:11 (2009)
4. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations (2012). URL http://cusp-library.googlecode.com. Version 0.3.0
5. Bertsekas, D.P.: Constrained Optimization and Lagrange Multiplier Methods. Athena Scientific (1996)
6. Bouaziz, S., Deuss, M., Schwartzburg, Y., Weise, T., Pauly, M.: Shape-Up: Shaping discrete geometry with projections. Computer Graphics Forum **31**(5), 1657–1667 (2012)
7. Boyd, S., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Foundations and Trends in Machine Learning **3**(1), 1–122 (2011)
8. Deng, B., Bouaziz, S., Deuss, M., Kaspar, A., Schwartzburg, Y., Pauly, M.: Interactive design exploration for constrained meshes. Submitted to Computer-Aided Design
9. Deng, B., Bouaziz, S., Deuss, M., Zhang, J., Schwartzburg, Y., Pauly, M.: Exploring local modifications for constrained meshes. Computer Graphics Forum **32**(2) (2013)
10. Deng, B., Pottmann, H., Wallner, J.: Functional webs for freeform architecture. Computer Graphics Forum **30**(5), 1369–1378 (2011)
11. Eigensatz, M., Kilian, M., Schiftner, A., Mitra, N.J., Pottmann, H., Pauly, M.: Paneling architectural freeform surfaces. ACM Trans. Graph. **29**(4) (2010)
12. Glaskowsky, P.N.: NVIDIA's Fermi: The First Complete GPU Computing Architecture (2009). URL http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf
13. Glymph, J., Shelden, D., Ceccato, C., Mussel, J., Schober, H.: A parametric strategy for free-form glass structures using quadrilateral planar facets. Automation in Construction **13**(2), 187–202 (2004). Conference of the Association for Computer Aided Design in Architecture
14. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010). URL http://eigen.tuxfamily.org
15. McAdams, A., Selle, A., Tamstorf, R., Teran, J., Sifakis, E.: Computing the singular value decomposition of 3×3 matrices with minimal branching and elementary floating point operations. Tech. rep., University of Wisconsin-Madison (2011)
16. NVIDIA: CUDA C Programming Guide. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/

17. NVIDIA: Fermi compute architecture whitepaper (2009). URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Version 1.1
18. NVIDIA: NVIDIA GeForce GTX 580 Datasheet (2010). URL http://www.geforce.com/Active/en_US/en_US/pdf/GTX-580-Web-Datasheet-Final.pdf
19. Poranne, R., Ovreiu, E., Gotsman, C.: Interactive planarization and optimization of 3d meshes. Computer Graphics Forum **32**(1), 152–163 (2013)
20. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing, 3rd edn. Cambridge University Press (2007)
21. Song, P., Fu, C.W., Goswami, P., Zheng, J., Mitra, N.J., Cohen-Or, D.: Reciprocal frame structures made easy. ACM Trans. Graph. **32**(4), 94:1–94:13 (2013)
22. Torres, Y., Gonzalez-Escribano, A., Llanos, D.: Understanding the impact of CUDA tuning techniques for Fermi. In: High Performance Computing and Simulation (HPCS), 2011 International Conference on, pp. 631–639 (2011)
23. Umeyama, S.: Least-squares estimation of transformation parameters between two point patterns. IEEE Transactions on Pattern Analysis and Machine Intelligence **13**(4), 376–380 (1991)
24. Vaxman, A.: Modeling polyhedral meshes with affine maps. Computer Graphics Forum **31**(5), 1647–1656 (2012)
25. Yang, Y.L., Yang, Y.J., Pottmann, H., Mitra, N.J.: Shape space exploration of constrained meshes. ACM Trans. Graph. **30**(6), 124:1–124:12 (2011)
26. Zhao, X., Tang, C.C., Yang, Y.L., Pottmann, H., Mitra, N.J.: Intuitive design exploration of constrained meshes. In: Advances in Architectural Geometry 2012