DGP 2019 PROJECT: DEFORMATION AND SIMULATION

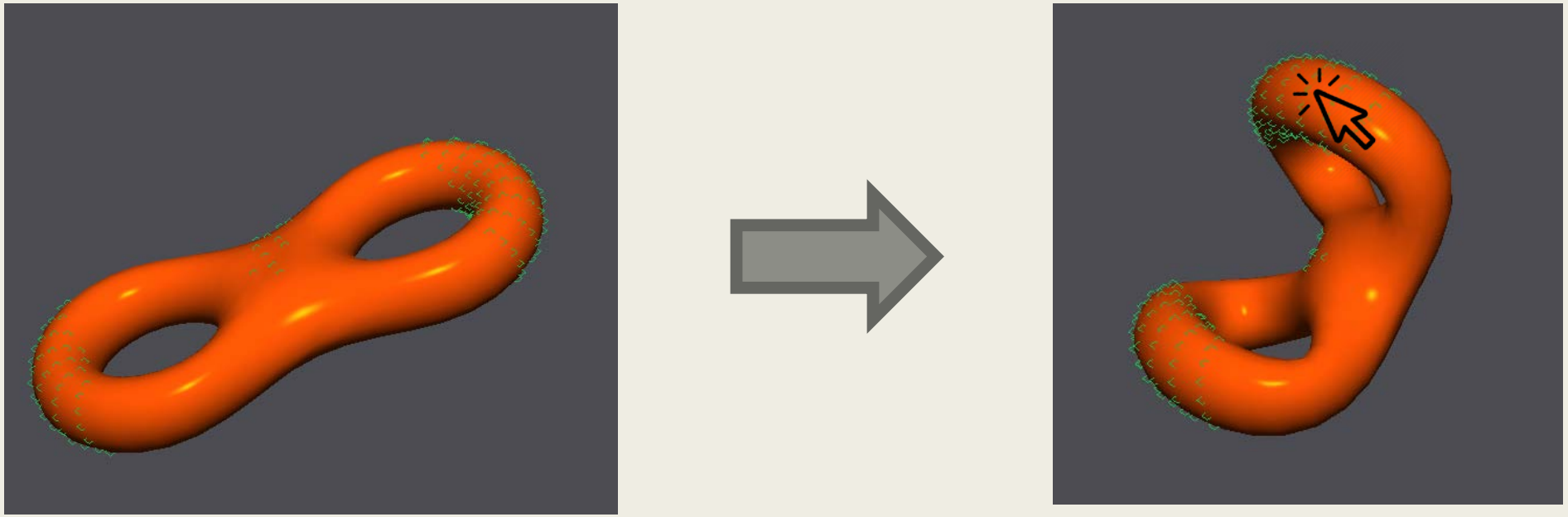
PART I: ShapeUp Deformation Code Framework

Intro

- **Your assignment for this week will be to go through these slides and to perform some tasks that come up on the slides**
- In the following, we introduce a code framework that implements **shape deformation using constraints** (ShapeUp)
- Later, this framework will be extended to **simulate** various types of objects
- In your **projects**, you will **extend this framework** in directions of your own choosing
- This is not a graded assignment, but it is extremely important to familiarize yourself with the code framework for the upcoming project
- Sometimes, the notation used here differs a little bit from that of Lecture 9: Constraint Based Modeling. This is because we follow the notation of the second paper (Projective Dynamics), which will be discussed next week.
- You can already work in the groups that you will form this week for the projects!

Shape Deformation with Constraints

- In the lectures, you have learned about ShapeUp. The following is a more technical introduction, focused on the implementation.
- Here, we are also **restricted to triangle meshes**, which was not the case in the lecture!
- Our rough goal is **to develop a tool that allows us to deform a shape in intuitive ways**, e.g. by imposing position constraints on some part of the mesh:



Shape Deformation with Constraints

- This type of shape editing can be expressed as a trade-off between constraints:

- Some constraints that want to keep the original shape
  - *Preserving edge lengths or*
  - *Preserving mean curvature or*
  - *Preserving triangle shape*
  - *…*
- And some constraints that want to change the shape
  - *Prescribing the positions of some vertices*
  - *Trying to achieve more smoothness*
  - *Trying to achieve more regular triangles*
  - *…*

Shape Deformation with Constraints

- This type of shape editing can be expressed as a trade-off between constraints:

- Some constraints that want to keep the original shape
- And some constraints that want to change the shape

- To formalize these constraints and their trade-off, we denote by $\mathbf{q} \in \mathbf{R}^{n\times 3}$ the positions of the vertices (each row denotes the x, y and z coordinates of one vertex)
- Constraints are defined locally, i.e. on a single vertex, triangle, edge, …
- To pick out the coordinates of the vertices that are needed to compute the constraint, we can use a "selection matrix" $\mathbf{S} = (0, \dots, 0, 1, 0, \dots, 0, 1, 0, \dots)$, which is everywhere 0 except at the indices of the vertices involved in the constraint, where it is 1
- If we let $\tilde{\mathbf{q}} = \mathbf{Sq}$, then $\tilde{\mathbf{q}}$ is a small $k \times 3$ matrix containing only the coordinates of the k vertices involved in the constraint

Shape Deformation with Constraints

- For a constraint, there is always also a ***constraint manifold*** and a ***constraint projection***, which is best explained on an example:
  - *If we want to retain the original length ℓ of an edge ("**Edge spring constraints**"), our constraints express this in the following way:*
  - *"The coordinates $\mathbf{q}_1$ and $\mathbf{q}_2$ of the vertices on the edge should be chosen such that the edge is in the set of edges whose length is ℓ."*
  - *Here, the constraint manifold C is "the set of edges whose length is ℓ" and the constraint projection $\mathbf{p}$ finds the closest edge $\mathbf{e}$, to the current edge $\mathbf{q}_1 - \mathbf{q}_2$, that is within the constraint manifold.*
  - *In formulas: $\mathbf{p}(\mathbf{q}) = \mathbf{min}_{\mathbf{e} \in C} ||\mathbf{ASq} - \mathbf{e}||^2$, where $\mathbf{S}$ is the selection matrix that picks $\mathbf{q}_1$ and $\mathbf{q}_2$ from all coordinates $\mathbf{q}$, and $\mathbf{A}$ computes the edge from these coordinates (by simply taking their difference)*
  - *Finally, the importance of a constraint can be tuned by a weight w*
- In the following, we will give an index to each constraint, such that a constraint is defined by the matrices $\mathbf{A}_i$ and $\mathbf{S}_i$, the constraint manifold C_i, the constraint projection $\mathbf{p}_i$ and the weight w_i

What exactly are the matrices $\mathbf{A}_i$ and $\mathbf{S}_i$ for the edge constraints?

Shape Deformation with Constraints

- **NOTE:** The Edge Spring constraints introduced here are slightly different from the Edge Length constraint introduced in the lecture, Slide 40:
  - In the lecture, we **directly prescribe target positions** for both vertices involved in the edge (note that there is mean-centering involved as well)
  - Here, we **prescribe the difference of the positions** of the vertices, but not directly their positions!
  - This is exactly the role of the matrix $\mathbf{S}_i\mathbf{A}_i$, which computes one edge from all vertex positions, and applies the constraint to this edge, instead of the vertex positions
  - This also circumvents mean-centering the vertex positions of each constraint, since usually the computed quantities are **translation-invariant**
- Make sure to understand this general principle: we usually do not constrain vertex positions directly, but rather quantities that are computed from the positions

Shape Deformation with Constraints

- An example for a constraint that tries to change the shape would simply be a constraint that wants to fix a single vertex to a (user-)specified position:
  - $\mathbf{S}_i$ *selects a single vertex* $\mathbf{q}_j$ *whose position we want to specify*
  - $\mathbf{A}_i = 1$
  - $C_i = \{\text{a single point } \mathbf{y} \text{ specified by the user}\}$
  - $\mathbf{p}_i(\mathbf{q}) = \mathbf{min}_{\mathbf{z}\in C}\left|\left|\mathbf{ASq} - \mathbf{z}\right|\right|^{\mathbf{2}} = \mathbf{min}_{\mathbf{z}\in C}\left|\left|\mathbf{q_j} - \mathbf{z}\right|\right|^{\mathbf{2}} \equiv \mathbf{y}$

Shape Deformation with Constraints

- **Recall:** Shape editing can be expressed as a trade-off between constraints:
  - *Some constraints that want to keep the original shape*
  - *And some constraints that want to change the shape*

- **When we choose a lot of these constraints, not all of them can be satisfied exactly, so we need to find a trade-off**
- We will do this by finding new positions $\mathbf{q}$, that minimize the cost of violating the constraints, which is defined as the sum (over all constraints) of the distance of the current quantities $\mathbf{A}_i\mathbf{S}_i\mathbf{q}$ to their constraint projections $\mathbf{p}(\mathbf{q})$:

$$E(\mathbf{q}) = \sum_{i} w_i \left|\left|\mathbf{A}_i\mathbf{S}_i\mathbf{q} - \mathbf{p}_i(\mathbf{q})\right|\right|^2$$

Recall: for $\mathbf{A}_i\mathbf{S}_i = \mathbf{Id}$, this is the proximity function ϕ from the lecture! Is it clear, why this cost function is a meaningful measure?

Shape Deformation with Constraints

- To efficiently compute a minimizer

$$\mathbf{q}_{\text{optimal}} = \min_{\mathbf{q}} E(\mathbf{q}) = \min_{\mathbf{q}} \sum_{i} w_i \left|\left|\mathbf{A}_i\mathbf{S}_i\mathbf{q} - \mathbf{p}_i(\mathbf{q})\right|\right|^2$$

we will split the minimization of this ***non-linear*** functional into two simple steps,

which we iterate until we reach a satisfyingly low cost E:

1. First, we compute all constraint projections $\mathbf{p}_i^* = \mathbf{p}_i(\mathbf{q}) = \mathbf{min}_{\mathbf{p}\in C}\left|\left|\mathbf{ASq} - \mathbf{p}\right|\right|^2$ Local Step
2. Then, pretending the constraint projections are now fixed (and do not depend on the positions), we can find optimal positions by solving the quadratic problem

$$\mathbf{q}^* = \min_{\mathbf{q}} \sum_{i} w_i \left|\left|\mathbf{A}_i\mathbf{S}_i\mathbf{q} - \mathbf{p}_i^*\right|\right|^2$$

which is equivalent to setting the gradient of the right hand side to zero, which amounts to solving the linear system

Global Step

$$\left(\sum_{i} w_i \ \mathbf{S}_i^{\mathbf{T}}\mathbf{A}_i^{\mathbf{T}}\mathbf{A}_i\mathbf{S}_i\right)\mathbf{q}^* = \sum_{i} w_i \ \mathbf{S}_i^{\mathbf{T}}\mathbf{A}_i^{\mathbf{T}}\mathbf{p}_i^*$$

Since the positions are stored in a $n \times 3$ matrix,here we actually see three linear systems, one for each coordinate (x, y and z).
Can you derive these linear systems? Start by computing the gradient of the cost function and setting it to 0.

Shape Deformation with Constraints: Implementation

- If implemented efficiently (compute projections in parallel, use sparse matrix for linear system), this is **fast** and can be used for **interactive shape deformation**
- To implement such a shape deformation framework, we need Code for
  - *The constraints: selection matrices and constraint projections*
  - *The linear system:*
    - Forming the left hand side (sparse matrix from the constraint's matrices)
    - Forming the right hand side (multiply "stacked" constraint projections by sparse matrix)
  - *Setup: load a mesh, choose initial constraints*
  - *Display mesh: OpenGL viewer, update vertex positions in viewer when mesh changes*
  - *Interaction: change constraint weights, move position constraints, add more constraints*
- The following slides will provide an overview over the code of the framework that implements the constraint based shape deformation ("***ShapeUp***")

ShapeUp Mesh Deformation – Constraints and Local-Global algorithm

"Local-Global algorithm" -> `ProjDyn::Simulator`

`<projdyn/projdyn.h>`

- For ShapeUp, assume that the boolean flag `m_dynamicMode` is **`false!`**
- Positions $\mathbf{q}$ -> represented as $n \times 3$ Eigen matrix `m_positions`
- The deformable mesh is completely described by these positions, and a matrix containing the vertex indices of all triangles `m_triangles`
- The simulator has a list of **constraints** -> `m_constraints`, `addConstraint()`, `addConstraints()`
- Everytime the constraints change, the left hand side matrix $\sum_i w_i \mathbf{C}_i^\mathbf{T} \mathbf{C}_i$ has to be built and factorized -> `initializeSystem()`
- Once the system is initialized, we can use the function `step()` to compute the optimized positions of the deformed mesh:
  - Iterate num_iterations times:
    - **Local step:** compute all constraint projections $\mathbf{p}$: `constraint->project(…)`
    - **Global step**:
      - First build right hand side: `m_rhs = m_constraint_mat_t * m_constraint_projection`
      - Then solve the three linear systems, corresponding to the x, y and z coordinates of the positions: `m_positions.col(coord) = m_solver.solve(m_rhs.col(coord));`

Constraints -> `ProjDyn::Constraint`

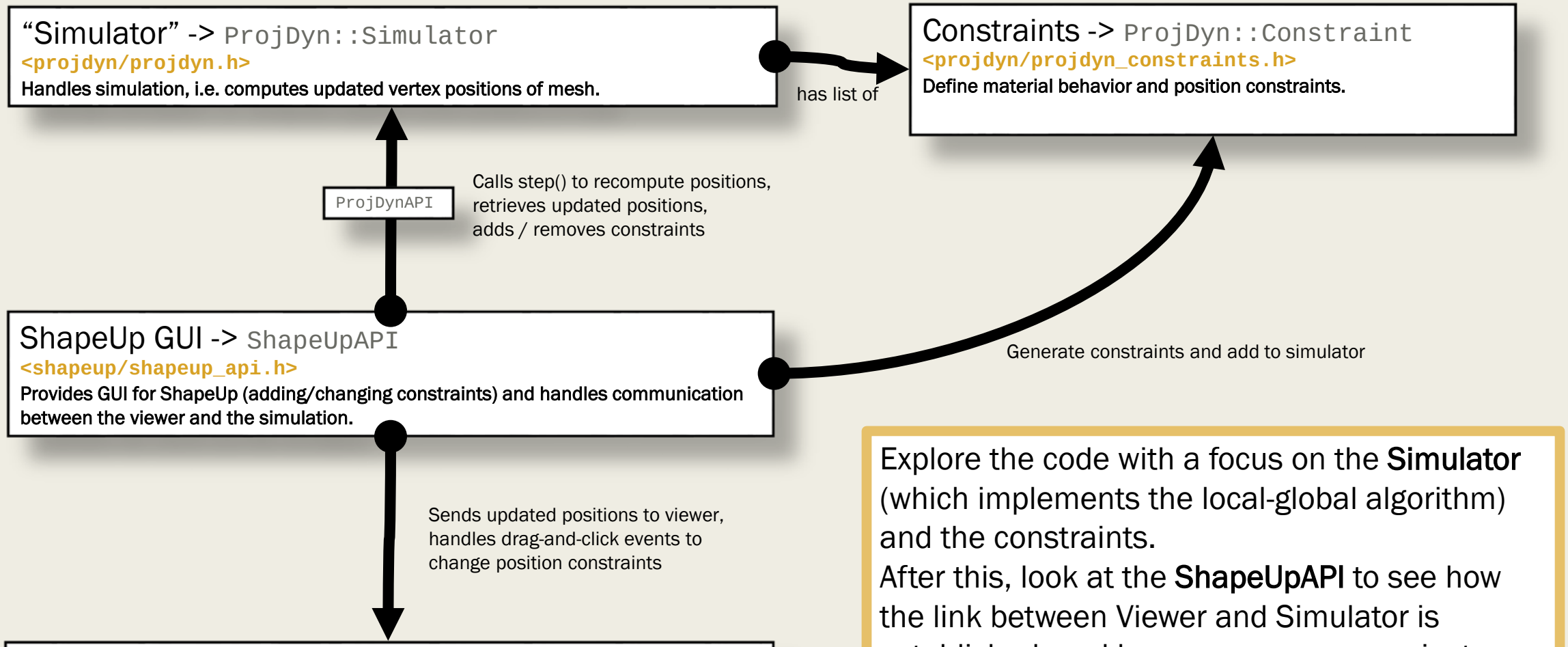
`<projdyn/projdyn_constraints.h>`

- Weight w_i -> `m_weight`
- Matrix $\mathbf{C}_i = \mathbf{S}_i \mathbf{A}_i$ -> given as Triplets of a sparse matrix `getTriplets()`
- Always assume $\mathbf{B}_i = \mathbf{Id}$ and does not need to be stored
- Constraint projection $\mathbf{p}_i = \min_\mathbf{p} \frac{w_i}{2} ||\mathbf{C}_i \mathbf{q} - \mathbf{p}||$ -> computed by function `project(positions, out_projections)`

- **Notes:**
  - Just like the vertex positions $\mathbf{q}$, the constraints projections $\mathbf{p}_i$ are stored as $k_i \times 3$ matrices, where k_i is the size of the constraint (e.g. 1 for edge springs, 1 for triangle bending, 2 for triangle strain). This size is returned by the method `getNumConstraintRows()`
  - All constraint projections are stacked into a large $k_{\text{total}} \times 3$ matrix, which appears on the right hand side of the global step. To find the position of a constraint in this list, we store the row of its first entry as `m_constraint_id`. The `project()` method directly writes the result of the constraint projection into the constraint matrix, using this value.

Why does "ProjDyn", "projective dynamics" and the word simulation keep popping up in slides and code?
What is the meaning of the boolean `m_dynamicMode`?
Wait for the next lectures and tutorials where this mystery will be unveiled!

ShapeUp Mesh Deformation – GUI and Interaction



Explore the code with a focus on the **Simulator** (which implements the local-global algorithm) and the constraints.
After this, look at the **ShapeUpAPI** to see how the link between Viewer and Simulator is established, and how one can communicate with the Simulator from outside and create constraints.
Note that ShapeUpAPI only communicates with Simulator through the **ProjDynAPI.**

Some general notes on the Code

- As briefly mentioned, the main class that implements the local-global algorithm is “ProjDyn::Simulator”, which has its name because it will later be used to simulate elastic deformable bodies (this will be discussed in future lectures and tutorials)
- To communicate with this class one can use a simplified API called ShapeUpAPI, which again communicates through ProjDynAPI. Both offer simple methods to set-up and run shape-deformation and simulations
- Constraints in the Simulator class are grouped into ConstraintGroups, which is simply a vector of Constraints, along with an additional weight multiplier (that gets multiplied to the individual weights) and a name for the group
  - *This is often helpful to change constraints that belong together in a unified way, and to have easier communication with the constraints from the outside*

Constraints

- Let us look at the implementation of constraints. The basis is an *abstract* class `Constraint`, from which specific constraints (edge springs, positions constraints, …) will be derived:

```
class Constraint {
public:
    Constraint(const std::vector<Index>& vertex_indices, Scalar weight) {
        m_vertex_indices = vertex_indices;
        m_weight = (std::max)(PROJDYN_MIN_WEIGHT, weight);
    }

    virtual void project(const Positions& positions, Positions& projections) = 0;

    void addConstraint(std::vector<Triplet>& triplets, Index& currentRow, bool
sqrtWeight = false, bool transpose = false) { … }

    Index numIndices() const { return m_vertex_indices.size(); }

    const std::vector<Index>& getIndices() const { return m_vertex_indices; }

    Scalar getWeight() const { return m_weight * m_weight_mult; }

    void setWeight(Scalar weight) { m_weight = weight; }

    void setWeightMultiplier(Scalar mult) { m_weight_mult = mult; }

    virtual ConstraintPtr copy() = 0;

protected:
    virtual std::vector<Triplet> getTriplets(Index currentRow) = 0;

    virtual Index getNumConstraintRows() = 0;

    std::vector<Index> m_vertex_indices;
    Scalar m_weight = 1.;
    Scalar m_weight_mult = 1.;
    Index m_constraint_id = 0;
};
```

Whenever you want to implement a new constraint, the methods `project` (compute the constraint projection) and `getTriplets` (return the entries of the sparse matrix $\mathbf{C}_i = \mathbf{A}_i\mathbf{S}_i$) need to be implemented.

Example – Edge Spring Constraints

- Here are the important methods `project` and `getTriplets` for the `EdgeSpringConstraints` class, which implements the edge spring constraints discussed earlier:

```
virtual void project(const Positions& positions, Positions& projection) override {
        // Check for correct size of the projection auxiliary variable;
        assert(projection.rows() > m_constraint_id);
        // Compute the current edge
        projection.row(m_constraint_id) = positions.row(m_vertex_indices[1]) - positions.row(m_vertex_indices[0]);
        // Rescale to rest length and put into constraint projection vector
        projection.row(m_constraint_id) /= projection.row(m_constraint_id).norm();
        projection.row(m_constraint_id) *= m_rest_length;
    }
```

```
virtual std::vector<Triplet> getTriplets(Index currentRow) override {
        std::vector<Triplet> triplets;
        // The matrix C_i = A_i S_i is very simple for edge springs:
        triplets.push_back(Triplet(currentRow, m_vertex_indices[0], -1));
        triplets.push_back(Triplet(currentRow, m_vertex_indices[1], 1));
        return triplets;
    }
```

Make sure you understand the input/output of these methods!

Example – Edge Spring Constraints

- This is almost all that needs to be implemented when implementing a new constraint.
- What remains is to implement
  - *a constructor (which in this case simply computes and stores the original length of the edge)*
  - *and two simple auxiliary methods*
    - `getNumConstraintRows()`, which returns the number of rows that a constraint projection requires (this is 1 for edge springs, since the constraint projection is an edge that is stored in a 1×3 matrix)
    - and `copy()`, which constructs a copy of the constraint and returns a it as a shared pointer

Example – Position Group Constraints

- Position group constraints – very similar to the position constraints introduced before, but instead specifying the positions of a group of vertices, insead of just one

```
virtual void project(const Positions& positions, Positions& projection) override {
        // Check for correct size of the projection auxiliary variable;
        assert(projection.rows() >= m_constraint_id + getNumConstraintRows());
        // Set given positions for selected vertices
        for (Index loc_v_ind = 0; loc_v_ind < m_vertex_indices.size(); loc_v_ind++) {
            projection.row(m_constraint_id + loc_v_ind) = m_target_positions.row(loc_v_ind);
        }
    }
```

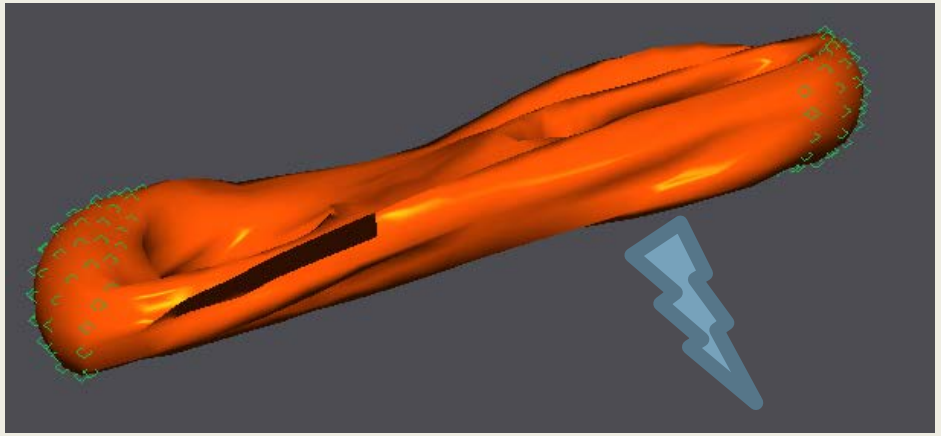
```
virtual std::vector<Triplet> getTriplets(Index currentRow) override {
        std::vector<Triplet> triplets;
        Index row = 0;
        for (Index v_ind : m_vertex_indices) {
            triplets.push_back(Triplet(currentRow + row, v_ind, 1.));
            row++;
        }
        return triplets;
    }
```

Try the application!

- Compile the Code in the usual way (using CMake to generate project files and then make or your favorite IDE to compile), the target/project name is "**shapeup**"
- Unless you are looking for bugs, always run this in **Release** mode
- Run the `shapeup` executable and follow these instructions
  - Open a mesh.
  - Vertices can be selected using **ctrl + left-click + drag**, which shows a selection box.
  - Selected vertices (blue) can be made into position constraint groups using the button **"Fix Selection"**. Create two such groups on different places on the mesh.
  - Vertices marked in green can be moved (as a group) by **alt/option + left-click + drag**, which changes the target positions of the constraints.
  - The shape is maintained by using either edge springs or triangle strain (or, once implemented, Iso 1-Ring constraints), which can be selected using the corresponding radio button. Note how much nicer the shape is maintained using the triangle strain + bending constraints, see the next slides for details.
  - Additional constraints can be added to selected vertices by the buttons to the lower left. For now, only smoothing constraints are available.
  - The currently used constraints are shown as groups in the window to the top right. There, the constraint weights can be changed using the sliders. Hovering over a slider, shows the vertices on which these constraints are defined.

Better shape constraints

- When using the simple edge-spring constraints to maintain the shape, we get quite ugly results for larger deformations:



- By using what we learned about shape and curvature, we can implement much better shape constraints:
  - *Triangle strain constraint – tries to maintain the shape of the original triangles, by restricting the **deformation gradient** (i.e. the linearized motion of the triangle) to be a rotation matrix*
  - *Bending constraint – tries to maintain the original mean curvature by restricting the mean curvature normal of each vertex to retain its original length*

Recall how the mean curvature normal is computed, and try to understand the code of the `BendingConstraint` based on this.

Isometric 1-Ring Constraints

- In the lecture you saw examples where yet another shape constraint was used: “Isometric 1 Ring Cells”, see Lecture 09, Slides 47 and 52
- These constraints are very good at recovering the original shape, but can not differentiate between bending and stretching (like the triangle strain and bending constraints from the previous slide)

Implement the “Isometric 1-Ring Cell” constraints!
(See next slide for details!)

Isometric 1-Ring Constraints

- Isometric 1-Ring Cell constraints are defined via the following procedure
  - *Select the vertices of a 1-Ring (i.e., the vertices around a specific vertex), using the matrix* $\mathbf{S}_i$
  - *From these, compute all* k *edges of the 1-Ring, using the matrix* $\mathbf{A}_i$*, let's call these edge* $e_1, \dots, e_k$
  - *For the constraint projection, we want to find the edges of a 1-Ring, that best fits the current 1-Ring, to the original 1-Ring (this means you will have to store the original edges of the 1-Ring)*
  - *See the next Slide for more info on the constraint projection*

Isometric 1-Ring Constraints

- To compute the constraint projection, you will need to find the best fit of the original 1-Ring edges, to the current (deformed) 1-Ring edges:
  - *Denote the original 1-Ring edges by* $\widehat{e_1}, \dots, \widehat{e_k}$
  - *Denote the current 1-Ring edges by* $e_1, \dots, e_k$
  - *The best fit of the current edges to the original edges, will simply be a rotation of the original edges. The matrix* $\mathbf{R}$ *performing the rotation can be found in the following way:*
    - Compute the matrix $\mathbf{E} = \sum_j \widehat{e_j} e_j^T$
    - Compute an SVD of $\mathbf{E}$, which gives you $\mathbf{E} = \mathbf{U\Sigma V^T}$
    - If the smallest singular value (the first entry on the diagonal of $\mathbf{\Sigma}$) is negative, revert the sign of the first column of U (i.e. multiply this column by -1)
    - Set $\mathbf{R} = \mathbf{VU^T}$
  - ***The SVD can be computed using Eigen:***

```
Eigen::JacobiSVD<Eigen::Matrix<Scalar, 3, 3>> svd(E,
Eigen::ComputeFullU | Eigen::ComputeFullV);
Eigen::Matrix<Scalar, 3, 3> U = svd.matrixU();
Eigen::Matrix<Scalar, 3, 3> V = svd.matrixV();
Scalar smallestSingularValue = svd.singularValues().coeff(0);
```

Isometric 1-Ring Constraints

- **A few more things to note on the Isometric 1-Ring Cell constraint:**
  - the quantity computed by $\mathbf{S}_i\mathbf{A}_i\mathbf{q}$ should be a $k \times 3$ matrix containing the k edges of the 1-Ring for the vertex on which the constraint is defined
  - Likewise, the quantity computed by the constraint projection $\mathbf{p}_i(\mathbf{q})$ is a $k \times 3$ matrix containing the best rigid fit of the original 1-Ring edges to the current 1-Ring edges (which are the rotated original edges, where the computation of the rotation matrix is described on the previous page)
  - NOTE: this constraint's size is k, the number of 1-Ring edges, which is equal to the number of neighbours. In particular, this number ***might be different for each vertex on which the constraint is defined***
- **To implement this constraint, fill in the empty methods of the class `IsometricOneRing` at the bottom of projdyn_constraints.h**
- To use this constraint, select it in the GUI as the active shape constraint. Unless implemented correctly, there will be an error in the command line, when trying to use these constraints.

What now?

- Familiarize yourself with the code, but also the application itself, i.e. play around with the current constraints and various shapes.
- Try to find interesting meshes to edit, and try to set goals for editing them. Not everything you will want to do works. Maybe this already gives you some ideas for extensions that you could do in the project?
- If you don’t understand something, write down your questions. The lectures go into more detail to explain the method, and in the next tutorials we will recap the code and answer your questions.
- This framework is new and was developed for this year’s DGP project. Don’t hesitate to report bugs by posting to the exercise forum!