Pyramid Coordinates for Deformation with Collision Handling

Wei-Chin Lin* Nafees Bin Zafar† Edwin Ng Jun Zhou
Oriental DreamWorks

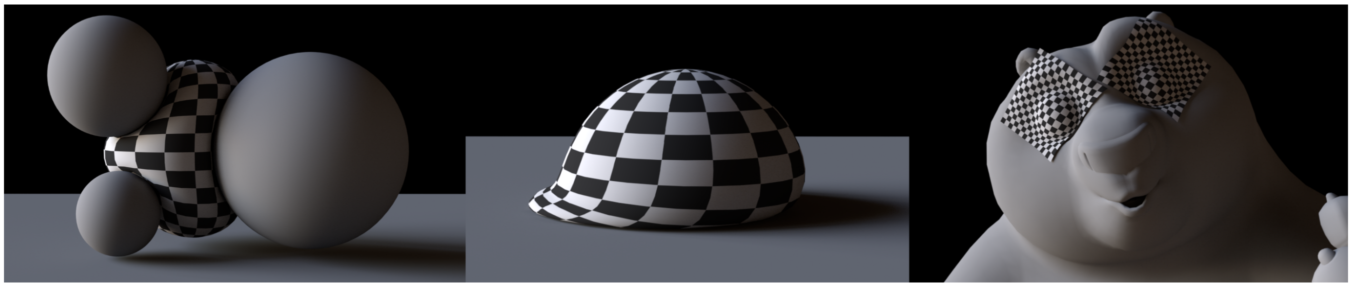

Figure 1: *The pyramid coordinate constraint and the proposed two-pass collision handling produce various collision-aware deformations (left and right). Plausible friction responses can be incorporated in the proposed method (middle).*

Abstract

We present an efficient implementation of the reconstruction of pyramid coordinates which are used for the deformation of animated characters. By reformulating the pyramid coordinates as an optimization problem with one-ring neighborhood constraints, we can solve the problem using an efficient projective solver. This greatly improves the overall performance, and makes it easier to incorporate other geometric constraints. Collisions between the deformed and kinematic geometries are handled using a two-pass methodology. By resolving collisions before applying pyramid coordinate constraints, we obtain a consistent result after the constraint projection. Dynamic simulation is also possible by modeling proper constraints and projection operators under the same framework.

Keywords: animation, rigging, deformation, dynamics

Concepts: •**Computing methodologies → Animation;**

1 Introduction

Pyramid coordinates [Sheffer and Kraevoy 2004] are very useful for modeling the deformed state soft tissues, such as the skin and the flesh, of CG characters. We refer readers to the open source *StretchMesh* plugin for Maya, which is based on a Jacobi implementation of pyramid coordinates [Str]. Typically coordinates are computed and stored at bind time. An iterative procedure is then used to reconstruct the mesh in the deformed configuration. In animation production, the art direction usually requires other constraints, such as attraction or volume preservation, to be enforced on the same character. This complicates the rigging process if the various constraints and operators are applied sequentially. In our implementation, we employ a unified optimization framework based on constraints as described in [Bouaziz et al. 2012; Bouaziz et al. 2014] for the pyramid coordinate reconstruction. The result is a geometry processing solver which is flexible for future extensions.

The simplest way to handle collisions between the deformed and other kinematic geometries is to find the vertices which are inside the collision surface, and project them onto the nearest surface points. Although this method is history-independent, it requires manifold geometry or heuristics to determine penetration. Moreover, the nearest point projection may cause inconsistent results since the nearest points might change over time. Another way is to detect the collisions and resolve them after the constraint solve. Although this method is robust, it effectively removes the soft properties of the pyramid coordinates. If we formulate the vertex-triangle collision as a constraint described in [Bender et al. 2015] and solve them (pyramid coordinates and collisions) altogether, there might be penetrations since the projective solver can only handle soft constraints. We present a history-based, two-pass algorithm which resolves all collisions after the constraint solve while preserving the properties of other soft constraints.

2 Pyramid Coordinates Constraint Solver

The pyramid coordinate reconstruction procedure can be reformulated as minimizing the shape proximity function of the input vertices $\mathbf{V} = \{\mathbf{v}_1, ..., \mathbf{v}_n\}$, $\mathbf{v}_1, ...\mathbf{v}_n \in R^3$,

$$\Phi(\mathbf{V}) = \sum_{i=1}^{m} w_i \|\mathbf{V}_i - P_i(\mathbf{V}_i)\|_2^2, \tag{1}$$

where w_i are non-negative weights that control the relative importance of the constraints, $\mathbf{V}_i \subseteq \mathbf{V}$ is the n_i vertices involved in shape constraint C_i (n_i is equal to one in the case of a pyramid coordinate constraint) and $P_i(\cdot)$ is the projection onto the constraint set C_i.

Similar to [Bouaziz et al. 2012], we employ an iterative two-step strategy to solve the minimization problem. In the first step, a projection $P_i(\mathbf{x})$ of a vertex $\mathbf{x}$ is computed while keeping $\mathbf{V}$ fixed. Each vertex $\mathbf{x}$ is then updated in the second step by minimizing Equation (1), keeping $P_i(\mathbf{x})$ fixed. Note that in the case of the pyramid coordinate constraint only, we can replace the second step by assigning

*gene.lin@odw.com.cn
†nafees.bin.zafar@odw.com.cn

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4282-7/16/07
DOI: http://dx.doi.org/10.1145/2897839.2927400

each vertex as its projection without solving a linear system. However, to incorporate other constraints that introduce a global solve, forming the matrices and solving the linear system is required. Algorithm 1 describes the nonlinear pyramid coordinate projection $P_i(\mathbf{x})$ for each vertex $\mathbf{x}$, which is equivalent to an iteration of the reconstruction procedure described in [Sheffer and Kraevoy 2004] (Section 2.2).

Reformulating the reconstruction process of the pyramid coordinates as constraint projection results in performance improvements since the projection of each vertex and the linear solve for each coordinate can be executed in parallel.

Algorithm 1 Pyramid Coordinate Projection

```
Require: $one - ring\ neighbor\ vertices\ \mathbf{n}_1, ..., \mathbf{n}_n$
1: procedure PYRAMIDCOORDPROJECT($\mathbf{x}$)
2:     $\mathbf{N} \leftarrow$ ComputeNormal($\mathbf{x}, \mathbf{n}_1, ..., \mathbf{n}_n$)
3:     for all the neighbor vertices $i$ do
4:         $\mathbf{n}_i \leftarrow$ ProjectNeighborsOnTheProjectionPlane($\mathbf{n}_i$)
5:     $\mathbf{x} \leftarrow$ ComputeMeanValueWeightedPosition($\mathbf{n}_1, ..., \mathbf{n}_n$)
6:     $\mathbf{x} \leftarrow$ OffsetNewPosition($\mathbf{x}, \mathbf{N}$)
```

3 Collision Handling

Similar to the collision handling in dynamic simulation of deformable solids, the typical approach to handling collision is to use history to decide whether nearby geometries have interpenetrated. If we apply the history-based approach after the pyramid coordinate constraint solve, we could have inconsistent results under the same collision configuration. A key observation is that there might be large numbers of arbitrary upstream geometric operations prior to the pyramid coordinate constraints, and the problem can be greatly simplified if the input vertices of the constraint solver are all intersection free. Therefore we apply a two-pass collision handling to the pyramid coordinate constraint solver (See figure 2). In the first pass, prior to the solve, we resolve the collisions on the input vertices based on their path over time. Continuous collision, and proximity detection are used to ensure collision-resolution for fast moving objects. We model the friction in the first pass using the position-based approach as described in [Bender et al. 2015].

The second pass of our collision handling algorithm is a combination of a series of collision detections and resolutions embeded in the constraint projection. After each iteration of solving Equation (1), the constraint solver is halted and we use the positions of each vertex between two iterations to detect and resolve the interpenetrations. This retains the soft properties of the pyramid coordinates.

Our collision handling only generates reasonable results if the resolved vertices are in the vicinity of the input vertices, this is as expected since the velocity of each vertex is not taken into account to determine its position. In some cases where interpenetrations are caused by serious overlapping of the collision objects, the pinched vertices may be handled by soft collision constraints rather than hard ones. A more sophisticated failsafe [Baraff et al. 2003] can also be applied to control the behavior of the pinched vertices.

4 Adding Dynamic Simulation

As shown by Bouaziz et al. [2014] it is very easy to include dynamic simualtion under the same framework. In our implementation, we employ exactly the same constraint projection operator for dynamic pyramid coordinate constraints. We add a closeness constraint on each vertex, targeting the position of the input vertex

Algorithm 2 Two-pass Collision Handling

```
1: $1st\ pass$ :
2: for all the vertices $i$ do
3:     loop collisionStep
4:         ComputeSearchDirection($\mathbf{x}_i$, $\mathbf{x}_i^{target}$)
5:         InterpolateCollisionMesh()
6:         if DetectCollision() then $\mathbf{x}_i \leftarrow$ ResolveCollision($\mathbf{x}_i$)
7: $2nd\ pass$ :
8: loop solverIteration
9:     for all the constraints $i$ do
10:        ConstraintProject($\mathbf{C}_i$)
11:    SolveLinearSystem()
12:    for all the vertices $i$ do
13:        if DetectCollision() then $\mathbf{x}_i \leftarrow$ ResolveCollision($\mathbf{x}_i$)
```

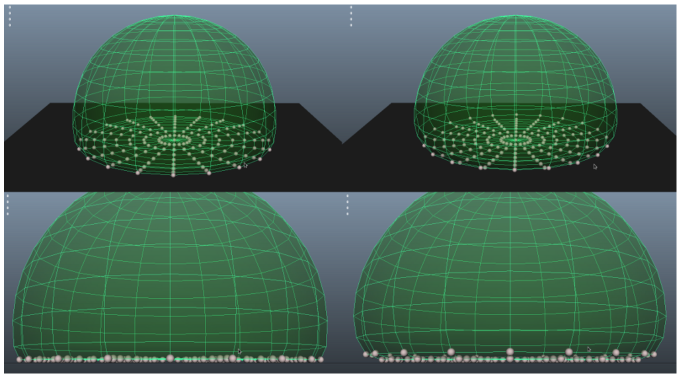

Figure 2: *Our two-pass collision handling resolves penetrations (small spheres in white) in the first pass (left). The second pass is executed after each constraint projection, retaining the soft property of the pyramid coordinates (right).*

which can come from animated input geometry. We can simulate various material properties on different areas of an animated character with different user painted weights for the closeness constraints.

References

BARAFF, D., WITKIN, A., AND KASS, M. 2003. Untangling cloth. In *ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, SIGGRAPH '03, 862–870.

BENDER, J., MÜLLER, M., AND MACKLIN, M. 2015. Position-based simulation methods in computer graphics. In *EUROGRAPHICS 2015 Tutorials*, Eurographics Association.

BOUAZIZ, S., DEUSS, M., SCHWARTZBURG, Y., WEISE, T., AND PAULY, M. 2012. Shape-Up: Shaping Discrete Geometry with Projections. In *Computer Graphics Forum*, Wiley-Blackwell, Hoboken, vol. 31, 1657–1667.

BOUAZIZ, S., MARTIN, S., LIU, T., KAVAN, L., AND PAULY, M. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans. Graph. 33*, 4 (July), 154:1–154:11.

SHEFFER, A., AND KRAEVOY, V. 2004. Pyramid coordinates for morphing and deformation. In *3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on*, 68–75.

StretchMesh. https://code.google.com/archive/p/stretchmesh/. Accessed: 2016-02-02.