Example-based Plastic Deformation of Rigid Bodies

Seminar: Current Topics in Physically-Based Animation

Younes Müller

January 20, 2017

Contents

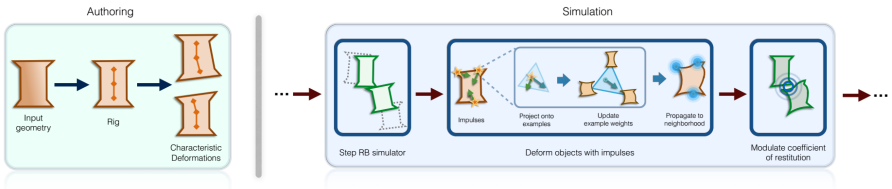

Figure 1: Overview of the authoring and simulation process.[BJB16]

1 Introduction

In modern days, computer graphics gains a lot of importance. David Kriesel once stated in one of his talks that the eye is the only broad band connection to the brain [Kri]. So the eye is the way to go to provide much detailed information to humans. Despite the large computational effort the results need to be computed nearly in real time for games and VR applications.
The field of physically based animation tightly belongs to computer graphics. A method to efficiently simulate the physical movement of objects is rigid body simulation. It assumes that the objects' shape is unchangeable This results in very fast computations. As the name suggests deformation of objects is not simulated during rigid body simulation In this seminar work I will describe and explain a method to solve this problem, described by Ben Jones, Niles Thuerey, Tamar Shinar and Adam W. Bargteil in their article "Example-based Plastic Deformation of Rigid Bodies."[BJB16] They developed a method, that extends a rigid body simulator. It is based on deformation example poses. A set of example deformations are provided by an artist or a soft body simulation. Additionally to the speed of this method it gives the user the ability to control exactly, how the deformed bodies look.

2 Method

2.1 Overview

Figure 1 Gives an overview over the method. During the authoring, the artist provides a set of example poses with the help of bones. The simulation step is done each time the rigid body simulator detects a collision. It hands the collision impulses to our routine. We then calculate a combination of the given examples that matches the deformation best. Then the combination is propagated towards the nearby vertices. To increase physical correctness, the non elastic part of the collision is increased.

2.2 Example

In Figure 2 we have a mesh and one example pose. The third image visualizes which vertex positions can be reached through blending of the example and the mesh.
Figure 3: The barrel collides with another object. Now the impulse $\boldsymbol{j}_i$ is acting upon

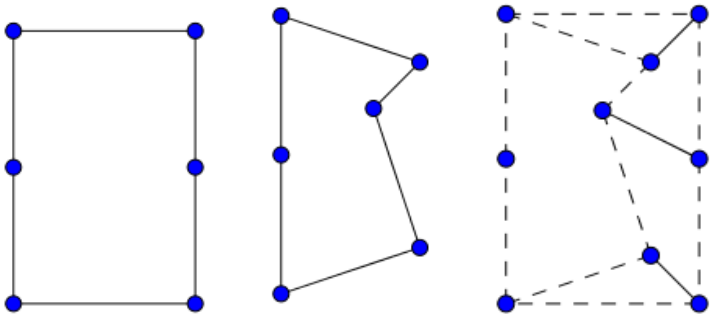

Figure 2: 1. the mesh 2. the example pose 3. the example space

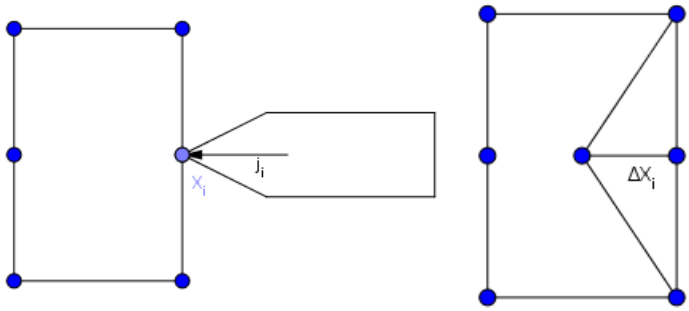

Figure 3: 1. the impulse j 2. the initial change

vertex $\boldsymbol{x_i}$. An ideal change in position $\boldsymbol{\Delta x}$ is calculated.

Figure 4: The ideal change $\boldsymbol{\Delta x}$ moves the vertex $\boldsymbol{x}$ out of the example space. So an example weighting is calculated, which matches the position best. Afterwards the change is propagated to the nearby vertices. The higher the distance to the hit vertex $\boldsymbol{x}$, the less impact the change has on the vertex.

2.3 Rigid Bodies

[Bar97] To simulate the translational motion of a rigid body it is sufficient to simulate the motion of the center of mass. As the vertices stay motionless relative to the center the position of every vertex can be calculated easily from the position of the center. If the motion is arbitrary it can still be represented by just a translational and a rotational movement. The simulation is done by a rigid body simulator, which needs the following parameters:

- densitiy ρ and mass m
- the moment of inertia, which is (in the bodie's frame of reference) constant to each body and behaves to rotational movement, as mass behaves to translational movement.

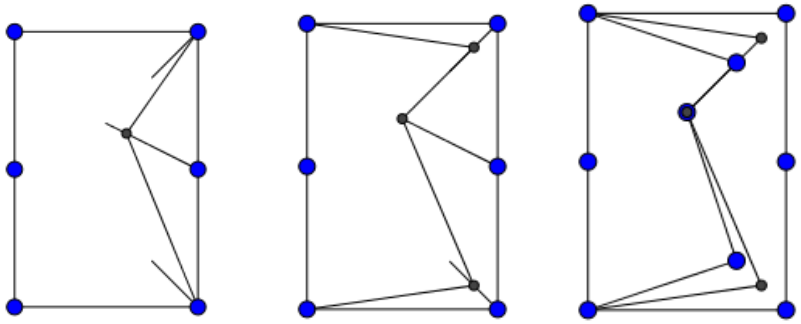

Figure 4: 1. approximated $\boldsymbol{\Delta x}$ in example space 2. the propagated change in example space 3. mesh, example and blend

- linear position $\boldsymbol{x} \in \mathbb{R}^3$ and momentum $\boldsymbol{p} \in \mathbb{R}^3$
- orientation $\boldsymbol{\Omega} \in \mathbb{R}^3$ and angular momentum $\boldsymbol{L} \in \mathbb{R}^3$
- coefficient of restitution C_r, which stores the ratio between elastic and non elastic collision

2.4 Skinning

It is very costy to provide physically correct positions for every single vertex. So a method named skinning is used. So called bones are added to the mesh (rigging) and every vertex is assigned to one or more bones. And similarly to real bones the vertices follow the movement of the bone they are attached to. To prevent artifacts a vertex can not only be assigned to one bone, but to multiple bones with weighting.
The bones behave like rigid bodies they can be moved and rotated, but they do not change their form. For example: In a model of an arm, the bones would be placed like to the bones in a real arm and if the bones move, the vertices follow. A vertex at the elbow would be associated with the bone of the upper and of the lower arm, and it's movement is a linear combination of the movement of the bones. So the vertex follows the movement of both bones. This gives an artist the opportunity to modify a mesh very easily.
For a mesh with N vertices and B bones some more values have to be stored.

- undeformed mesh vertex positions: $\boldsymbol{u} \in \mathbb{R}^{N\times 3}$
- skinning weights: $\boldsymbol{W} \in \mathbb{R}^{N\times B}$
- the desired transformation, which consist of:

  rotation: $\boldsymbol{R} \in \mathbb{R}^{B\times 3\times 3}$

  translation: $\boldsymbol{T} \in \mathbb{R}^{B\times 3}$

The skinning weight matrix $\boldsymbol{W}$ contains a row for each vertex and a column for each bone. The value W_{ib} represents how much percent of the movement of bone b affects vertex $\boldsymbol{x}$.
So to get the transformed position of a single vertex we apply the weighted transformation of each bone on the vertex.

$$\boldsymbol{x}_i = \sum_{b \in B} \boldsymbol{W}_{ib} \left(\boldsymbol{T}_b + \boldsymbol{R}_b \boldsymbol{u}_i \right) \tag{1}$$

As we want to use E example poses on every vertex we don't have one desired Transformation, but one for every pose, generated by the artist. So we extend $\boldsymbol{R}$ and $\boldsymbol{T}$ by one dimension for the examples and add a matrix that contains, how much every example pose acts on every vertex:

- $\boldsymbol{R} \in \mathbb{R}^{B \times E \times 3 \times 3}$
- $\boldsymbol{T} \in \mathbb{R}^{B \times E \times 3}$
- $\boldsymbol{E} \in \mathbb{R}^{N \times E}$

We have to somehow consider mutliple example poses. The intuuitive approach is to use linear interpolation and weight the individual transformations with the weighting matrix $\boldsymbol{E}$. We now can determine the new position of each vertex by:

$$\boldsymbol{x}_i = \sum_{b \in B} \boldsymbol{W}_{ib} \left(\sum_{e=1}^{E} \boldsymbol{E}_{ie} \left(\boldsymbol{T}_b + \boldsymbol{R}_{be} \boldsymbol{u}_i \right) \right) \tag{2}$$

With the translational part this works fairly well, however linear interpolation does not work well on rotation matrices. This is due to the fact, that a 3 dimensional rotation has 3 degrees of freedom, while a rotation matrix has 9 entries. So 6 Entries are somehow redundant and have to be bound via constraints (orhogonality and determinant is 1). So if we linearly interpolate each single entry of a roation matrix it is not guaranteed, that the resulting matrix is itself a rotation matrix. Quaternion rotation provides a solution to this.

2.4.1 Quaternion rotation and QLERP

Ken Shoemake presented a method to use quarternions for a 3 dimensional rotation instead of a matrix [Sho85]. Quaternions are an extension of the complex numbers. They have 3 imaginary units and are based on the identity

$$i^2 = j^2 = k^2 = ijk = -1 \tag{3}$$

Each quaternion has the form: $a + bi + cj + dk$. Standard algebraic operations on quaternions can be determined by using the standard form. For example addition:

$$\boldsymbol{p} = a + bi + cj + dk \tag{4}$$
$$\boldsymbol{q} = x + yi + zj + wk \tag{5}$$
$$\boldsymbol{p} + \boldsymbol{q} = (a + x) + (b + y)i + (c + z)j + (d + w)k \tag{6}$$

The length is:

$$||\boldsymbol{p}|| = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{7}$$

How exactly a quaternion can be used to rotate a vector can be read in [Sho85].
Quaternion rotation has a main advantage over matrix rotation. Having 3 degrees of freedom in a 3D rotation and 4 entries in a quaternion there is only one additional entry, instead of 6. So only one constraint is needed to provide a valid rotation. It turns out that a rotation quaternion has to be of length one, a constraint that is easy to fulfill.
So to interpolate between two rotations it is sufficient to linearly interpolate between the two rotation quaternions and normalize them afterwards. [Kr05] The formula for interpolation of the rotation quaternions $\boldsymbol{q}$ and $\boldsymbol{p}$ with the parameter $t \in [0, 1]$ the following formula is used:

$$l(t; \boldsymbol{p}, \boldsymbol{q}) = \frac{(1-t)\boldsymbol{p} + t\boldsymbol{q}}{||(1-t)\boldsymbol{p} + t\boldsymbol{q}||} \tag{8}$$

However it should be noted that the interpolation provides no constant speed, considering the distance on the rotation circle. As seen in Figure 5 a change at the beginning or end moves the point E on the circle further than at the middle. This effect however is small and therefore disregarded, as the exact solution, provided in [Sho85] would be far more computational complex [Kr05]. To blend multiple rotations we now use a matrix, containing the quaternion values for each example pose $\boldsymbol{R}^{4\times E}$ and the weightings $\boldsymbol{E}^{N\times E}$. This works, because the columns of $\boldsymbol{E}$ sum up to 1.

$$\boldsymbol{R}'_i = \text{QLERP}(\boldsymbol{E}_i, \boldsymbol{R}) = \text{normalize}\left(\sum_{e\in E} \boldsymbol{E}_{ie}\boldsymbol{R}_e\right) \tag{9}$$

2.5 Recap Skinning

Now we have a method to deform meshes with the help of bones and to blend in given example poses. The input consists of the constant data:

- The Vertex positions in undeformed state: $\boldsymbol{u} \in \mathbb{R}^{N\times 3}$
- The weighting of each bone for the vertcies: $\boldsymbol{W} \in \mathbb{R}^{N\times B}$
- The transformation for each example pose and bone, consisting of
  The rotational part $\boldsymbol{R} \in \mathbb{R}^{B\times 4\times E}$
  The translational part $\boldsymbol{T} \in \mathbb{R}^{B\times 3\times E}$

and the changing data:

- example Weights $\boldsymbol{E} \in \mathbb{R}^{N\times E}$

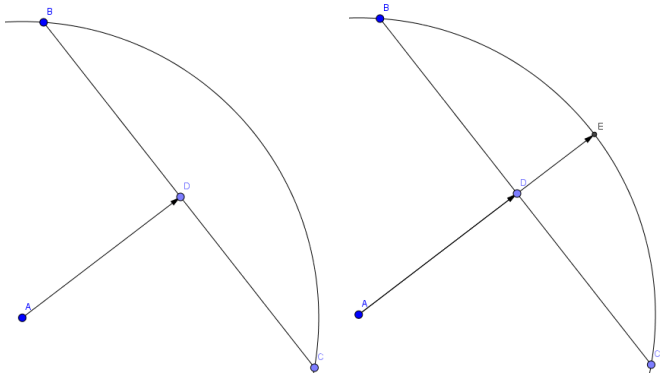

Figure 5: On the left: Idea, how linear interpolation between rotation matrices looks. If linear interpolation between the two points B and C on a circle is used, the resulting point does not lie on the circle. On the right: Interpolating with quaternions. The result can be easily mapped onto the sphere by normalizing the quaternion.

Let rotate$(\boldsymbol{q}, \boldsymbol{x})$ be the function that rotates the position $\boldsymbol{x}$ by the quaternion $\boldsymbol{q}$. The translational part is interpolated linearly, the rotational by the $QLERP$-algorithm. The skinned mesh vertex positions $\boldsymbol{x} \in \mathbb{R}^{N\times 3}$ can now be calculated by:

$$\boldsymbol{x}_i = \sum_{b\in B} \boldsymbol{W}_{ib} \left(\sum_{e=1}^{E} \boldsymbol{E}_{ie}\boldsymbol{T}_{be} + \text{rotate}(\text{QLERP}(\boldsymbol{E}_i, \boldsymbol{R}_b), \boldsymbol{u}_i) \right) \tag{10}$$

2.6 Calculate example weighting

When two objects collide, the rigid body simulator passes the vertex $\boldsymbol{x}_i$, where the collision occurred and the impulse $\boldsymbol{j}_i$. The goal is now to calculate a useful change $\boldsymbol{\Delta E}$ of the example weights, that matches the impact. They are later applied to the example weight matrix $\boldsymbol{E}$. Initially every entry of the first column of $\boldsymbol{E}$ contains 1 and the rest is 0. The first column represents the weighting of the undeformed mesh. That means the mesh is undeformed at the beginning. As example poses are used, not every deformation is possible. So we first calculate an ideal deformation for the vertex being hit. Then we solve a minimization problem, trying to get good example weightings, so that the distance between the ideal deformation and the skinned position of the vertex is minimal. Afterwards we propagate this change to the vertices around.

2.6.1 Projection

We want to compute the desired position based on the impulse, given to us by the rigid body simulator. Impulse is a change in momentum. So we can write:

$$\boldsymbol{j} = m\boldsymbol{v} \tag{11}$$

To get the change in position of the object in the next timestep, we divide by the mass and multiply by the duration of the timeslice:

$$\boldsymbol{\Delta x} = \frac{\Delta t}{m}\boldsymbol{j} \tag{12}$$

To prevent deformation on contact and on soft impacts we add a threshold β, which prevents the object from deforming to easily and takes out some of the energy. We end up with the following formula for the impacted vertex:

$$\boldsymbol{\Delta x}_i = \frac{\Delta t}{m}\max(||\boldsymbol{j}_i|| - \beta, 0)\frac{\boldsymbol{j}_i}{||\boldsymbol{j}_i||} \tag{13}$$

We now construct the Jacobian matrix $\boldsymbol{J}$ by differentiating formula 10. This is done with help of automatic differentiation [SB]. The matrix contains all partial derivatives of

$$\boldsymbol{J}_i = \frac{\partial \boldsymbol{x}_i}{\partial \boldsymbol{E}_{ie}} \in \mathbb{R}^{3\times E} \tag{14}$$

$$= \begin{pmatrix} \frac{\partial \boldsymbol{x}_{i1}}{\partial \boldsymbol{E}_{i1}} & \cdots & \frac{\partial \boldsymbol{x}_{i1}}{\partial \boldsymbol{E}_{in}} & \cdots & \frac{\partial \boldsymbol{x}_{i1}}{\partial \boldsymbol{E}_{iE}} \\ \frac{\partial \boldsymbol{x}_{i2}}{\partial \boldsymbol{E}_{i1}} & & \frac{\partial \boldsymbol{x}_{i2}}{\partial \boldsymbol{E}_{in}} & & \frac{\partial \boldsymbol{x}_{i2}}{\partial \boldsymbol{E}_{iE}} \\ \frac{\partial \boldsymbol{x}_{i3}}{\partial \boldsymbol{E}_{i1}} & \cdots & \frac{\partial \boldsymbol{x}_{i3}}{\partial \boldsymbol{E}_{in}} & \cdots & \frac{\partial \boldsymbol{x}_{i3}}{\partial \boldsymbol{E}_{iE}} \end{pmatrix}. \tag{15}$$

It contains the change in vertex position depending on a change in the example weighting. So given a change of position $\boldsymbol{\Delta x}$, in the example space we can compute the change of example weightings by:

$$\boldsymbol{\Delta e}_i = \boldsymbol{J}_i^T \boldsymbol{\Delta x}_i \tag{16}$$

$$\begin{pmatrix} \Delta e_{i1} \\ \cdots \\ \Delta e_{in} \\ \cdots \\ \Delta e_{iE} \end{pmatrix} = \begin{pmatrix} \frac{\partial \boldsymbol{x}_{i1}}{\partial \boldsymbol{E}_{i1}} & \frac{\partial \boldsymbol{x}_{i2}}{\partial \boldsymbol{E}_{i1}} & \frac{\partial \boldsymbol{x}_{i3}}{\partial \boldsymbol{E}_{i1}} \\ \cdots & & \cdots \\ \frac{\partial \boldsymbol{x}_{i1}}{\partial \boldsymbol{E}_{in}} & \frac{\partial \boldsymbol{x}_{i2}}{\partial \boldsymbol{E}_{in}} & \frac{\partial \boldsymbol{x}_{i3}}{\partial \boldsymbol{E}_{in}} \\ \cdots & & \cdots \\ \frac{\partial \boldsymbol{x}_{i1}}{\partial \boldsymbol{E}_{iE}} & \frac{\partial \boldsymbol{x}_{i2}}{\partial \boldsymbol{E}_{iE}} & \frac{\partial \boldsymbol{x}_{i3}}{\partial \boldsymbol{E}_{iE}} \end{pmatrix} \begin{pmatrix} \Delta x_{i1} \\ \Delta x_{i2} \\ \Delta x_{i3} \end{pmatrix} \tag{17}$$

This formula however requires the change to be in the space of possible changes, that can be achieved by the examples. Our goal is now to get a good value for the change in example weights $\boldsymbol{\Delta e}$. To achieve that we have to get the change, that happens, if we apply $\boldsymbol{\Delta e}$ as close as possible to the desired change $\boldsymbol{\Delta x_i}$. Let $\boldsymbol{x}_i(e)$ be the function that moves the vertex according to the example weighting e. Then we have to solve:

$$\boldsymbol{\Delta e} = \min_{\boldsymbol{\Delta \hat{e}}} ||(\boldsymbol{x}_i(\boldsymbol{e} + \boldsymbol{\Delta \hat{e}}) - \boldsymbol{x}_i(\boldsymbol{e})) - \boldsymbol{\Delta x}_i||^2 \tag{18}$$

A method to solve minimization problems is the method of steepest descent. It uses the fact that a gradient always points in the opposite direction of the steepest descent. So if we start at some point and follow the direction of the gradient in infinite many,

infinitesimal small steps we reach a local minimum, as far as it exists. Similar to a marble rolling around on a surface. It will sometime reach a lokal minimum, as it always follows the direction of steepest descent. Of course the marble has to be somewhat sticky, otherwise its momentum would prevent it from following the gradient directions. Instead of going infinite small steps we take finite many not so small steps. As we do not opt for physical accuracy it is sufficient to go one step. As initial value $\boldsymbol{\Delta e} = 0$ is used. This makes the first term of the minimization cancel out. The gradients can by definition be calculated by transposing the Jacobi matrix:

$$\boldsymbol{J}_i^T \boldsymbol{\Delta x}_i \tag{19}$$

is the gradient on Δx.

The stepwidth of the step is chosen, so that it is dependent on the impact and user controllable: $\alpha||\boldsymbol{\Delta x}||$ The approximated change in Example weightings is calculated by:

$$\boldsymbol{\Delta e}_i = \alpha||\boldsymbol{\Delta x}||\frac{\boldsymbol{J}_i^T \boldsymbol{\Delta x}_i}{||\boldsymbol{J}_i^T \boldsymbol{\Delta x}_i||} \tag{20}$$

2.7 Propagation

By now we only calculate a change of example weights for one vertex. In order for the result to look acceptable, the change has to be propagated nearby vertices. To calculate the distance between two vertices, Dijksta's algorithm is used. The change of Example weights for every vertex is saved in the Matrix $\boldsymbol{\Delta E}$. To propagate the change we modify each column by:

$$\boldsymbol{\Delta E}_j \mathrel{+}= \phi\left(\frac{\text{dist}(\boldsymbol{x}_j, \boldsymbol{x}_i)}{\gamma}\right)\boldsymbol{\Delta e}_i \tag{21}$$

$$\phi(x) = \begin{cases} 2x^3 - 3x^2 + 1 & : x < 1 \\ 0 & : otherwise \end{cases} \tag{22}$$

The function ϕ is called kernel smoother. It assures that the change is applied fully on the impacted vertex. With increasing distance the extend of application decreases. How far the changes should be propagated is determined by the value γ.

2.8 Application of the Deformation

To let the impact look more natural, the deforming does not happen instantaneously, but over time. The Matrix $\boldsymbol{\Delta E}$ contains the change in example weighting that still has to be applied. Every timestep a bit of the deformation accumulator $\boldsymbol{\Delta E}$ is applied and decreased by the same amount. This is achieved through the user given parameter λ.

$$\boldsymbol{E} \mathrel{+}= (1-\lambda)\boldsymbol{\Delta E} \tag{23}$$

$$\boldsymbol{\Delta E} \mathrel{*}= \lambda \tag{24}$$

Note that the amount of application is high directly after the impact and decreases over time. For $t \to \infty$ the whole change is applied.

2.9 Restitution Modifiaction

As a deformation consumes energy it is necessary to show this in the movement of the object. With the deformation the nonelastic part of the collision increases. The ratio between the elastic and non-elastic part, the coefficient of restitution is modified by:

$$C_r := \min(C_r^*, C_r + \mu\Delta t, \exp(-\nu||\boldsymbol{\Delta E}||_f)C_r^*) \tag{25}$$

The first term C_r^* is the coefficient that is initially used by the rigid body simulator. The second term increases over time to avoid jittering and the third parameter decreases over time. μ and μ are user given parameters.

2.10 Userparameters

The following parameters are adjustable by the user and determine:

- α: how much the impact deforms the bodies
- β: how strong the impact has to be to be noticed
- γ: how far the deformation is propagated
- λ: how long the deformation should take
- μ and ν: how elastic the collision is modified

2.11 Fracture

Fracturing of the mesh is supported. During design time the artist splits up the mesh into individual parts. The parts are hold together by constraints. If a force exceeds a threshold the contraints are removed and the model is fractured.

3 Discussion and Conclusion

3.1 Conclusion

The method described provides an efficient solution to plastic deformations. It is based on example deformations. So the user has intuitive control over how the deformed meshes look. It uses a rigid body simulator and only deformes the objects on impact. So the speed of rigid body simulators is combined with the ability to simulate plastic deformation. Experiments have shown that very few example poses are sufficient to get fairly realistic looking results. It is not necessary to provide hundreds of examples for each mesh.

3.2 Performance

Using the method to simulate deformation adds a avarage of 42% to 59% of the initial computing time to every timestep in the tests. It is significantly slower than regular rigid body simulation. One reason for this is the collision detection. There are very efficient alogrithms to calculate collisons on convex bodies. To use them on arbitrary meshes, the meshes are split up to convex parts. As the shape of the objects changes during the method the splitting cannot be precomputed. So the collision has to be detected for non convex bodies, which is less efficient.
Although the method is slower than usual rigid body simulation it still is extremely efficient, compared to other methods for computing plastic deformations as the finite element method or spring mass systems.

3.3 Future Work

A possible improvement would be automated rigging. Then example poses can be provided by soft body simulators and integrated into the system.
Implementing an application for modeling the objects or integrating a plugin for a given application would give the possitility to modify parameters with brushes individually for the vertices.

References

[Bar97] David Baraff. An introduction to physically based modeling: Rigid body simulation 1—unconstrained rigid body dynamics. University Lecture, 1997.

[BJB16] Tamar Shinar Ben Jones, Nils Thuerey and Adam W. Bargteil. Example-based plastic deformation of rigid bodies. *ACM Transactions on Graphics*, 35(4), July 2016.

[Kr05] Ladislav Kavan and Jiří Žára. Spherical blend skinning: A real-time deformation of articulated methods. *I3D '05*, July 2005.

[Kri] David Kriesel. Spiegelmining – reverse engineering von spiegel-online https://www.youtube.com/watch?v=dkfn_e9lfis

[SB] O. Stauning and C. Bendtsen. Fadbad++, flexible automatic differentiation using templates and operator overloading in c++ http://fadbad.com.

[Sho85] Ken Shoemake. Animating rotation with quaternion curves. *ACM SIGGRAPH Computer Graphics*, 19, November 1985.