Project Report on
MESHLESS DEFORMATION BASED ON SHAPE MATCHING

A SIGGRAPH 2005 paper by Matthias Muller, Bruno Heidelberger, Matthias Teschner and Markus Gross

COMPUTER GRAPHICS CSE 528
SRIKANT AGGARWAL (109890795)
STONY BROOK UNIVERSITY

Abstract

Requirement to simulate real–time soft body deformations in games and animation industry has attracted a lot of graphics research interest in the past decade. Shapeless matching is a major methodology used for this purpose. The project implements "Shapeless matching based on shape matching", a paper which simulates real-time deformations based on shape matching.

Ackowledgements

I am thankful to Dr. Hong Qin for helping me with deciding the project and for his valuable guidance and helping me with queries and doubts. I am also thankful to Computer Graphics TA, Max Curran, for his assistance.

Introduction

The project is based on implementation of "Meshless deformation based on shape matching", a paper by Mathias Muller, Bruno Heidelberger, Matthias Teschner, Markus Gross which was presented in the SIGGRAPH 2005.
The techniques used in simulation of deformable objects before the paper were either not suitable for real-time or were not robust. Also, these techniques were not controllable.
The paper by Muller aimed to make deformable object simulation feasible in games. It modeled elasticity based on Shape Matching technique. The approach followed in paper is simple to implement and efficient in terms of memory requirements and run-time. Also, the deformations can be controlled by a set of parameters.

Technical Background

The paper builds on Explicit Euler Integration scheme (which is fast) and tries to make it more stable.

Explicit Euler Integration
Explicit Euler Integration scheme modifies Newton's Equation of motion as follows :

$$v(t + \Delta t) = v(t) + F_{ext} * \Delta t / m$$
$$x(t + \Delta t) = x(t) + v(t) * \Delta t$$

The integration scheme is easy to implement and is quite fast. But as it tries to predict future events based on the current factors, this makes it unstable.
For a mass spring system, $F_{ext} = -k (l_0 - x(t))$. So modified equations (assuming that current velocity is rate of change in position) become:

$$v(t + \Delta t) = v(t) - k * (l_0 - x(t)) * \Delta t / m$$
$$x(t + \Delta t) = x(t) + v(t + \Delta t) * \Delta t$$

Also, suppose that t is the instant at which mass m was released. So $v(t) = 0$. Equations become

$$v(t + \Delta t) = -k * (l_0 - x(t)) * \Delta t / m$$
$$x (t + \Delta t) = x(t) - k * \Delta t^2 * (l_0 - x(t)) / m$$

Clearly, if the spring stiffness, k, or the time difference, Δt, is very large or the connected mass, m, is very small, then $| x(t + \Delta t) - l_0 | >> | x(t) - l_0 |$ resulting in a gain of potential energy. But as the system started with an initial kinetic energy of 0 (v(t) was 0), this means there is a gain in overall energy of the system hence making it unstable. The paper based on Shape Matching technique tries to make the system stable by introducing an additional factor to the calculation of velocity.

Shape Matching Technique

The object is animated as a system of point masses with only external forces acting on it. At every time step, the technique tries to bring particle's original position (x_i^0) to its current position (x_i) by applying a set of optimal transformations, thereby yielding its goal position (g_i).

Algorithm

The paper assumes that the object, represented as a set of points, reached current state by undergoing a set of transformations from its original state. After applying the Shape Matching technique to each point, it then gets its goal position. Say, for any point the current position is x_i, original position was x_i^0 and the goal position is g_i.

Then, $x_i = R(x_i^0 - t_0) + t$ (assuming only rotation and translation)

In order to get the goal position, g_i we try to minimize, $(R(x_i^0 - t_0) + t) - x_i$.

As each particle undergoes same transformations, we can solve for optimal R, t_0, t by using least square minimization technique.

Hence, minimizing $\Sigma m_i * (R(x_i^0 - t_0) + (t - x_i))^2$ gives

$t_0 = \Sigma m_i x_i^0 / \Sigma m_i = x_{cm}^0$

$t = \Sigma m_i x_i / \Sigma m_i = x_{cm}$

To solve for optimal R, the author solves for general transformation, A and then extracts the rotational part out it. The equation to be minimized becomes $\Sigma m_i * (Aq_i - p_i)^2$ where $q_i = (x_i^0 - x_{cm}^0)$ and $p_i = (x_i - x_{cm})$.

Solving, we get,

$A = A_{pq} * A_{qq}$ where $A_{pq} = \Sigma m_i p_i q_i^T$, $A_{qq} = (\Sigma m_i q_i q_i^T)^{-1}$

Now as A_{qq} is symmetric, the rotational part would only be contained in A_{pq}. R contained in A_{pq} is then found using polar decomposition algorithm.

Putting optimal values for t_o, t and R, we get the goal position, g_i.

$g_i = R(x_i^0 - x_{cm}^0) + x_{cm}$

Once the goal position for a particle has been found, the algorithm uses the goal position to add a restoring force and a tendency for the particle and hence the whole object to regain a state which can be defined by a set of valid transformations. This makes the system stable.

$v(t + \Delta t) = v(t) + F_{ext} * \Delta t / m + \alpha (g_i - x(t)) / \Delta t$

Next the paper discusses extension of the algorithm to various scenarios :

a. *Rigid Body Motion*

As a rigid body can undergo only rigid transformations (translation and rotation), putting $\alpha = 1$ in the restoring force factor, the case reduces to that of rigid body motion.

b. *Linear Deformation*

The paper extends its results to simulation of linear deformations. When simulating linear deformations effects of scaling and shearing needs to be inculcated. This the paper does, by first calculating R as a combination of transormation A and rotation R. Specifically new $R = \beta A + (1-\beta)R$.

c. *Quadratic Deformation*

The paper states that the results can be further extended to simulate quadratic deformations like shearing and twisting. To simulate quadratic deformations, we need to minimize $\Sigma m_i(A'q_i' - p_i)^2$ where $q_i' = [q_x, q_y, q_z, q_x^2, q_y^2, q_z^2, q_xq_y, q_yq_z, q_zq_x]$. Solving for A' gives $A' = A_{pq}' * A_{qq}'$ where $A_{pq}' = (\Sigma m_i p_i q_i'^T)$ and $A_{qq}' = (\Sigma m_i q_i' q_i'^T)$. R, a combination of transformation and rotation, would equal $R = \beta A' + (1-\beta)R$.

Design & Implementation

Design

main.h file - The main.h file declares the global variables which are used in the system.

Variables:

a. mode : The deformation type selected by the user
b. alpha, beta : Controllable parameters which can be adjusted as per user requirements
c. cube_color : The color of current cube displayed on screen
d. groundLevel : This contains the level at which ground is located
e. elasticity : Collision elasticity

main.cpp file - The main file or the starting point of the system.

Methods:

a. main : This is the starting point of application. It helps in creating window, setting display mode, initializing the system, specifying various callbacks for display, idle state etc.
b. initialize : The method initializes the system by setting up the viewport, loading textures, enabling depth test, setting the perspective view matrix, setting light properties, initializing ground level etc.
c. loadTextures : This method loads sky and sandy textures used in background.
d. idleFunc : The callback is called when the system is in idle state, that is no user interaction is taking place. It calls glutPostRedisplay method to update and render the system.
e. displayFunc : This is the callback where rendering takes place. The method renders the background, updates the object and render it on the screen. Pseudo code:

```
If the object array contains an object
    If the object has just been created
        Initialize the object (its original position etc.)
    else
        Do shape matching for the object
        Update the object's position
    Render the object
```

f. menuCallback : This callback method gets called when user selects any menu option. Based on the selection by user, the method initializes new object, defines alpha, beta, elasticity and calls glutPostRedisplay to render the object.
g. makeMenu : The method is used to create menu, the modes available like rigid body motion with and without rotation etc.

DeformableObject class : DeformableObject class models an object. It contains the methods necessary to handle loading of object, rendering of object, finding original center of mass, finding the current center of mass, calculating and updating each particle's goal position based on the type of deformation, moving the particles etc.

Methods:

a. initialize : The method loads the obj file, initializes the original position and rotation of the object (of particles contained in the object), calculate the original center of mass for the object , calculates quantity like Aqq etc. which only needs to be calculated once for the whole system (doesn't change at each time step). Pseudo code:

```
For each particle in the object
    Initialize its original position and current position
If mode if LINEAR
    Initialize Aqq
If mode is QUADRATIC
    Initialize tildeAqq
```

b. loadFile : The method accepts a string which is path of the obj file to be loaded. It then parse the obj file, extracts information about the particles in the object, normals and the triangles.
c. getOriginalCenterOfMass : It calculates the initial or orginal center of mass for the object.

d. getCenterOfMass : It calculates current center of mass for the object.
e. matchShape: The method based on the shape matching technique calculates the goal position for each particle in the object. For this it calculates the current center of mass of the object and an optimal R which is then applied to calculate the goal position. Pseudo code:

```
For each particle in the object
    Calculate goal position depending on mode
    If the particle can move
        Update its goal position
```

f. update : This method based on the goal position calculated for each particle in matchShape method updates the current position for the particles. Pseudo code:

```
For each particle in the object
    Update its velocity and position
```

g. draw : After the current position for each particle has been calculated the object is rendered in the draw method.

Variables:

a. particles : A vector of particles contained in the object.
b. normals : This is a vector of normals that are extracted from obj file and used for rendering purposes.
c. triangles : This is a vector of triangles that are extracted from the obj file and used in object rendering.
d. original_center_of_mass : Object's original or initial center of mass.
e. center_of_mass : Object's current center of mass.
f. Aqq, tildeAqq : These are the matrices used in the simulation of linear, quadratic deformations and needs to initialized just once (As they do not change at each time step).

Particle class : The class models a particle. Each particle has attributes like its mass, its radius, forces acting on it, its position etc.

Methods:

a. initialize : The method is used to set the original position of the particle (which is also equal to its current position at that time).
b. update : The method based on the particle's goal position updates its current velocity and current position. It also updates the velocity of the particle based on whether it is colliding with any other particle or whether its y level is less than the ground level of the system.

```
Update particle's velocity
Update particle's position
If the particle collides
    Update its velocity based on the colliding model
```

c. getOriginalPosition : Method returns original (initial) position of the particle
d. getTempPosition : This method returns the temporary position of the particle which acts as a precursor for calculating its goal position and is an addition to paper implementation to add stability to the system.
e. getPosition : This method returns the current position of the particle.
f. setGoalPosition : The method sets the goal position for the particle which is calculated using shape matching technique.
g. setTempPosition : This is the setter for particle's temporary position.
h. getVelocity : The method returns current velocity for the particle.
i. setVelocity : The method sets (updates) current velocity for the particle.
j. getForces : The method returns current forces acting on the particle.
k. getMass : The method returns mass of the particle.

l. canMove : Returns true or false depending on whether the particle will collide with other particle's or will its y level be below the ground level if it moves further.
m. getRadius : Returns the particle's radius.

Variables:

a. original_position : The variable stores the original position of the particle
b. goal_position : The variable stores the goal position for the particle.
c. temp_position : The variable stores the intermediate position for the particle.
d. position : The variable contains current position of the particle
e. velocity : The variable contains particle's current velocity
f. force : Vector representing the forces acting on the particle
g. mass, radius : Mass and radius of the particle

Vector class : This is a class modeling a Vector quantity.

Methods:

a. operator+ : The '+' operator is overloaded to add two vector quantities
b. operator- : Overloaded '-' operator for Vector class
c. operator* : Overloaded '*' operator for multiplication of a vector with a scalar.
d. operator/ : Overloaded '/' for division of a vector by a scalar.
e. magnitude : Returns the magnitude of the Vector
f. normalize : Returns unit vector in the vector's direction
g. project : Accepts another vector as an argument and returns its projection on the current vector

Variables:

a. x, y, z : Denotes x, y, z coordinates of the vector

Triangle class : This class represents a triangle.

Variables:

a. vertices : An int array containing index for the particle corresponding to each vertex (index to the particles vector in DeformableObject class).
b. normals : An int array containining index for normal to each vertex (index to the normal vector in DeformableObject class).

Bitmap class : The class implements Bitmap loader and expects the full bmp file name (along with path). It gives methods to access the image data, image width, image height etc. which can be used to initialize image textures. The class is written by Mark Bernard and is free to use.

Globals.h : The file externalizes global controllable constants. This file is being imported by various other classes. It also contains the enum DEFORMATION_TYPE for possible types of deformations.

Implementation

Deformation : Deformation of the object is simulated using the algorithm based on shapeless matching technique described in paper.

It was observed that the object simulation based on the paper is still unstable. So an intermediate temporary position has been introduced in the implementation. Say, the particle's initial position was x_i^{t-1}, its initial velocity was v_i^{t-1}, the temporary position is x_i^T, its temporary velocity is v_i^T, current center of mass is x_{cm}, the final position after current time step is x_i and final velocity is v_i. Then they are related by following set of equations :

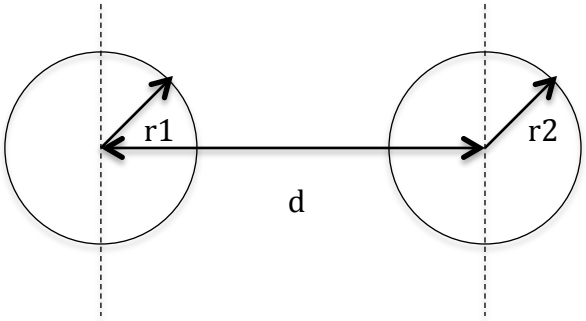
$$v_i^T = v_i^{t-1} + \Delta t * F_{ext} / m$$
$$x_i^T = x_i^{t-1} + v_i^T * \Delta t$$
$$x_{cm} = \Sigma m_i x_i^T / \Sigma m_i$$

$$g_i = Rq_i + x_{cm}$$
$$v_i = v_i^{t-1} + \alpha (g_i - x_i^T) / \Delta t + F_{ext} * \Delta t / m$$
$$x_i = x_i^{t-1} + v_i * \Delta t$$

Collision Detection : For each particle one of the following types of collision is possible:

a. Particle collides with ground : This is detected after each time step by comparing the updated particle's y position with the ground level. In case the particle's y position is less than the ground level, then the particles velocity is reflected in the y direction

b. Particle collides with another particle : For this each particle is assumed to a sphere with some radius. If the Euclidean distance between any particle and any other particle becomes less than the sum of radius of the two particles, velocity of the two particles is reflected along with normal at the point of intersection (vector connecting center of the two particles). Particle collides if d <= (r1+r2).



Requirements

The project is an XCode project.

Frameworks Used:

a. OpenGL Framework

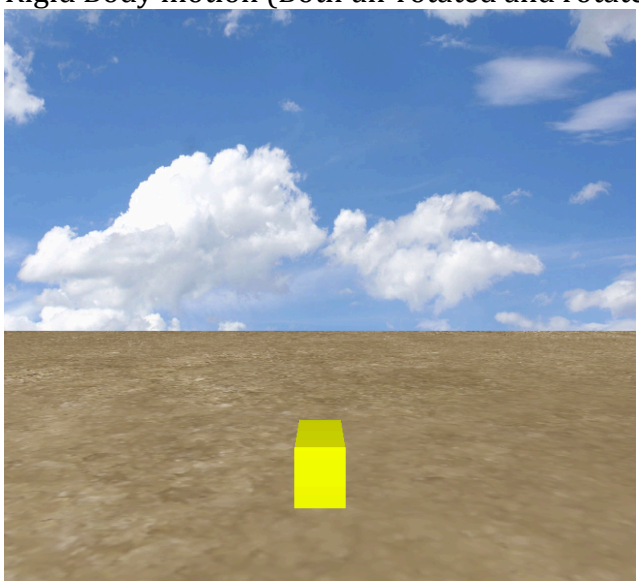
b. GLUT Framework

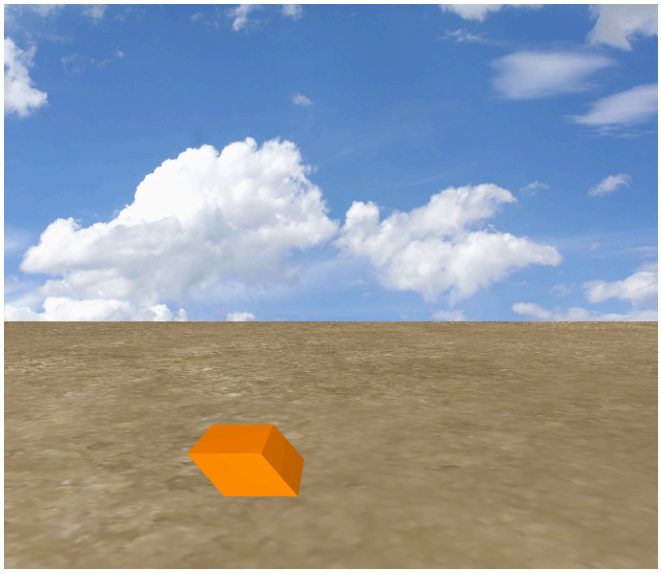
Library Used:

Eigen Library

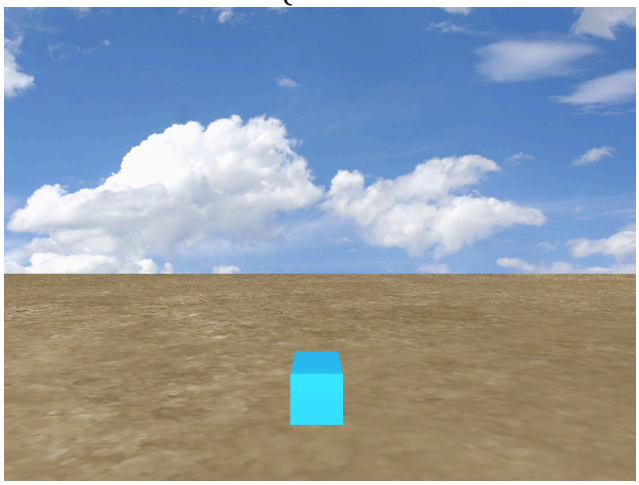
Results

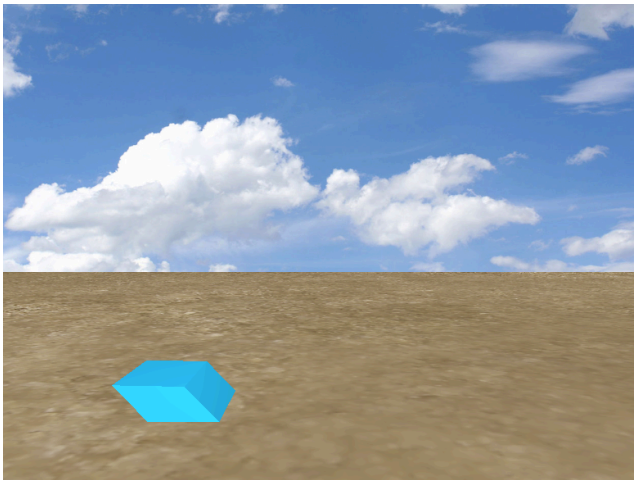
Rigid Body motion (Both un-rotated and rotated):





Linear Deformation (Both un-rotated and rotated)





Conclusion & Future Work

The project is able to successfully simulate rigid body motion and linear deformations both un-rotated and rotated in real time with little changes to the original algorithm described in paper (to bring stability). In quadratic deformation implementation it was found that the system behaves unnaturally and its deformation is not as expected. Further improvements to the algorithm described in paper needs to be done to ensure that the proper system behavior in case of quadratic deformations. Also, a better collision handling technique needs to be used.

References

[1] Muller, M., Heidelberger, B., and Gross, M. 2005. Meshless Deformations Based on Shape Matching. In Proceedings of ACM SIGGRAPH 2005, New York, USA, 471-478.

[2] Muller, M., Dorsey, J., Macmillan, L., Jagnow, R., and Cutler, B. 2002. Stable Real-Time Deformations. In Proceedings of ACM SIGGRAPH Symposium on Computer Animation 2002, 49-54.

[3] Muller, M., McMillan, L., Dorsey, J., and Jagnow, R. 2001. Real-time simulation of deformation and fracture of stiff materials. In Proceedings of Eurographics Workshops, Eurographics Association 2002, 99-111.