Sri Lanka Institute of Information Technology

Soft Body Dynamics

Project ID : 16-018

Project Proposal

B.Sc. Special (Honours) Degree in Information Technology

Submitted on 09/03/2016

Soft Body Dynamics

Project ID : 16-018

Authors:

Student ID	Name	Signature
IT 13 0013 08	M.Y. Alimudeen	
IT 13 1079 56	E.M.D.C.M. Ekanayake	
IT 13 0862 44	S.M.D.R. Kithsiri	
IT 13 1195 22	W.M.G.N.P. Kumara	

Supervisor

…….

Dr. Darshana Kasthurirathna

DECLARATION

We declare that this is our own work and this project proposal does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

…….	……..
M.Y. Alimudeen	E.M.D.C.M. Ekanayake
…….	…….
S.M.D.R. Kithsiri	W.M.G.N.P. Kumara

ABSTRACT

This document proposes a plugin for the Unity game engine to simulate 3-dimensional mesh deformation and fluid dynamics, in real time. The proposed system expects to use mass-springs model of simulation along with shape matching to implement deformable solids. The former method employs a physically based model which will favour accuracy, while the latter is based on a non-physically based model, which is fast to compute. Fluids will be simulated using a metaball approximation with the Unity game engine. Density, viscosity and surface tension-like properties will be modelled after the Navier-Stokes equations. Refraction and transparency for the metaballs and the other visual fluid properties will be achieved using specialised shaders. Accomplishing this would help independent unity developers who are unable to produce complicated deformable object systems in their games, as well as the open source community in an industry where most technologies are closed-source and proprietary.

TABLE OF CONTENTS

Contents

INTRODUCTION

Background

Video games are currently a multibillion dollar industry, with everyone from children to middle-aged people actively playing video games. Game development is done by companies with budgets exceeding a million dollars to small indie games with sometimes just one person working on the game.

Independent developers, however have become by far the biggest source of games and with the explosion of mobile computing, the independent development scene has only gotten bigger. Developers typically work on a game on top a game engine, like Unreal Engine.

Physics in game engines is generally computed the same way. Three-dimensional objects are given invisible colliders that typically bound the volume the mesh takes up. The colliders are used for collision detection and physics calculations. These objects can range from primitive cubes with 8 vertices to complex human models with several thousand vertices. These objects are considered rigid bodies, in the sense that the mesh doesn't undergo any change (besides the three main transformations) during the simulation.

The majority of all games exclusively use rigid bodies, with any mesh changes (animations) being scripted or pre-rendered for the scene. This is primarily because of how expensive it is to compute and simulate deformations in a mesh, like in a ball bouncing or metal sheet bending. Objects in which deformations can occur in simulations are considered soft bodies.

Current implementations of accurate real-time soft body simulation are closed source. No commercial game engine yet supports soft body dynamics, besides cloth and hair, which do not react to environmental collisions. There exist some plug-ins for engines which facilitate soft-body dynamics in this way, however they are expensive and closed source, making it harder for an indie developer to implement such a system in their game.

Similarly, fluids are another area in video games that engines either do not support or offer poor support for. Heavier engines (Cry and Unreal) provide an ocean tool that allows the creation of realistic looking water bodies, but these react poorly to the environment, often only responding to basic interaction, like floating objects. They do not typically respond visually, like creating waves. Games have skirted around this limitation by pre-rendering the fluid for known objects.

Fluid simulation is extremely expensive, so has not penetrated mobile gaming well. Current games with fluids being the core mechanic, exploit rigid particles to model the behaviour of fluids. These trade off fluid appearance, for semi-accurate behaviour.

Some existing physics simulations use the concept of metaballs to visualize electromagnetic fields between particles. Some independent developers have extended this concept of using metaballs to simulate linear uniform fields, to an approximation of fluids in other fields. A metaball approximation has been worked on by some developers for video games, but these still have huge limitations when it comes to fluid behaviour.

Literature Review

Fluid Simulation

The most realistic fluid simulations use a particle-based method [1] [5] which simulates fluids as a collection of particles that each have their own velocity, position and other properties. The system is able to handle fluid dynamics like waves, drag, create particle sprays and simulate Brownian motion. Complicated fluid phenomena like surface tension, diffusion and convection can also be simulated. This model uses the Navier-Stokes equations to accurately maintain the conservation of momentum, mass and energy of particles.

However, this method is computationally expensive [1] and even if resource consumption is reduced (to render real-time) the issue of handling the rest of a game's environments, collisions and meshes still exists.

An alternative to the particle based simulation, is using metaballs to render the curved surfaces of water particles [2]. Metaballs have been widely used because of its simplicity and applicability.

An implementation of metaballs uses the unity engine's own objects and colliders as an approximation to metaballs. A metaball is defined as a function of n-dimensions and a thresholding value [3]. Metaballs are generally used to represent physical volumes or spaces in areas such as magnetic or electric fields in physics and so on.

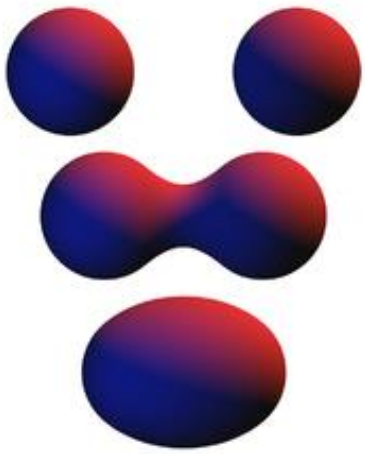


Figure 1 - Metaballs

Flocking algorithms [4], seem like an alternative to the Navier-Stokes equations for simulating a simple surface tension in fluids. Flocking is also simple and fast to compute.

Special shaders need to be written for the metaballs to resemble fluids such as water, steam, lava and so on. A shader is a computer program used for the production of colour within an image or a model [6] [7]. Different types of shaders have different applications. 3D shaders are run for each vertex given in a model and are used to map the 3D vertices to a 2D coordinate system. They can generate new vertices for the model and can further subdivide a model for smoothing and tessellation [8] [9].

Pixel shaders compute colour and other attributes for pixels. They can be used for translucency, shadows, specular highlights and other phenomena.

Mesh Deformation

The deformation of meshes through the application of physics has been discussed substantially since Terzopoulos' paper [9]. Several methods have since been proposed to address the issue, focusing on performance, accuracy and stability [10] [12] [16] [18]. Terzopoulos has attempted to simulate the action of every particle in a mesh individually, and their reaction after a collision.

His implementation uses partial differential equations to accurately model the collisions and the resulting deformations. However, the system does not prioritise performance over accuracy. They have also identified stability issues with the discrete equations used. Terzopoulos' method is now described as a finite element method of simulating soft bodies, which is a method that produces realistic results.

Another method proposed is a rigid body based deformation system, which simulates a soft body as a set of rigid bodies that are connected by a set of constraints [11]. Much of the work and research in this domain is closed-source.

Shape matching has been put forward as yet another method to simulate soft bodies [12] [16] [17]. It works without worrying about the connections between the vertices. Instead of trying to simulate how every particle in a mesh would react in a deformation, this method creates a target shape for a deformation and then integrates each vertex to its target location discretely with the time step. Polar decomposition is used to compute rotations.

Shape matching is described as a non-physically based simulation method. That is, one that isn't physically accurate. It does however require significantly less resources than physically based methods such as finite element simulation [13].

A mass-spring system is another physically based model that works by computing each particle or vertex as a mass and the connections between them as elastic springs, using an extended version of Hooke's Law [14] [15]. The model allows for one-dimension (rope-like), two-dimensional (cloth) as well as three-dimensional meshes. The application of Newton's laws of motions to the nodes yields us a system of differential equations which can then be used to calculate the deformation.

Research Gap & Research Problem

Most liquid simulation software uses the particle based simulation model to get accurate and realistic results for fluid renderings. These use complicated algorithms and procedures to calculate physical phenomena like dynamic ripples, buoyancy, animating waves and foam. These simulations can take from a few minutes to close to an hour to render a single frame.

Computing particle based fluids has been identified as resource intensive, yet realistic. Particle based methods also have to be explicitly parallelised. In video games, maintaining a minimal render time for the system is of utmost importance. Games have to be able to run at 60 frames per second, leaving only about 15 milliseconds to render a single frame. The particle based method for simulating fluids is therefore unsuitable for a video game environment, which requires real time rendering.

Commercial game engines provide full support for rigid bodies in games, handling movement, collisions and other interactions. When considering meshes and their interactions, a major contributor to the performance edge that engines have over more accurate simulation software, is due to the optimisation targeted at rigid bodies. This however, makes the game engine itself poor at handling soft bodies.

As a result of this, existing games rarely use deformable meshes, and ones that do, rely on proprietary systems by middleware vendors to achieve dynamic real-time deformations. These systems rely on the rigid-body based deformation systems, of which literature is mostly non-existent.

The discussed finite element simulation method, although accurate, takes too long to render in real time, so hasn't widely been used in video game environments.

The proposed shape matching method, is a non-physically based computation method for deformable solids, which means that it doesn't provide deformations accurately, prioritising frame render speed. This makes shape matching ideal for real-time rendering situations like in video games.

Mass-spring systems have been applied to games, but mostly exist for cloth systems, with a specialised version working for the Chainmail Algorithm.

OBJECTIVES

Main Objectives

Produce a plugin for a video game engine to handle soft body deformations. This plugin should be available to game developers for use in their games.

Produce another component in the plugin that can handle simple fluid dynamics in real time for a video game environment.

Specific Objectives

A metaball particle prefab has to be created to be used in the fluid system. This includes the script that controls the particles behaviour like surface tension and viscosity. The users of the plugin should be able to change some of the attributes to suit their game.

A library of shaders should be provided to the user of the plugin. These shaders would include lit and unlit shaders for several types of fluids, some of them incorporating transparency and refraction.

A fluid particle factory (or tap) will be produced for developers to generate fluid particles in-game. The factory should itself be able to produce particles in different ways that games may require, with exposed functions to configure duration, rate of emission, pulsing emission and so on.

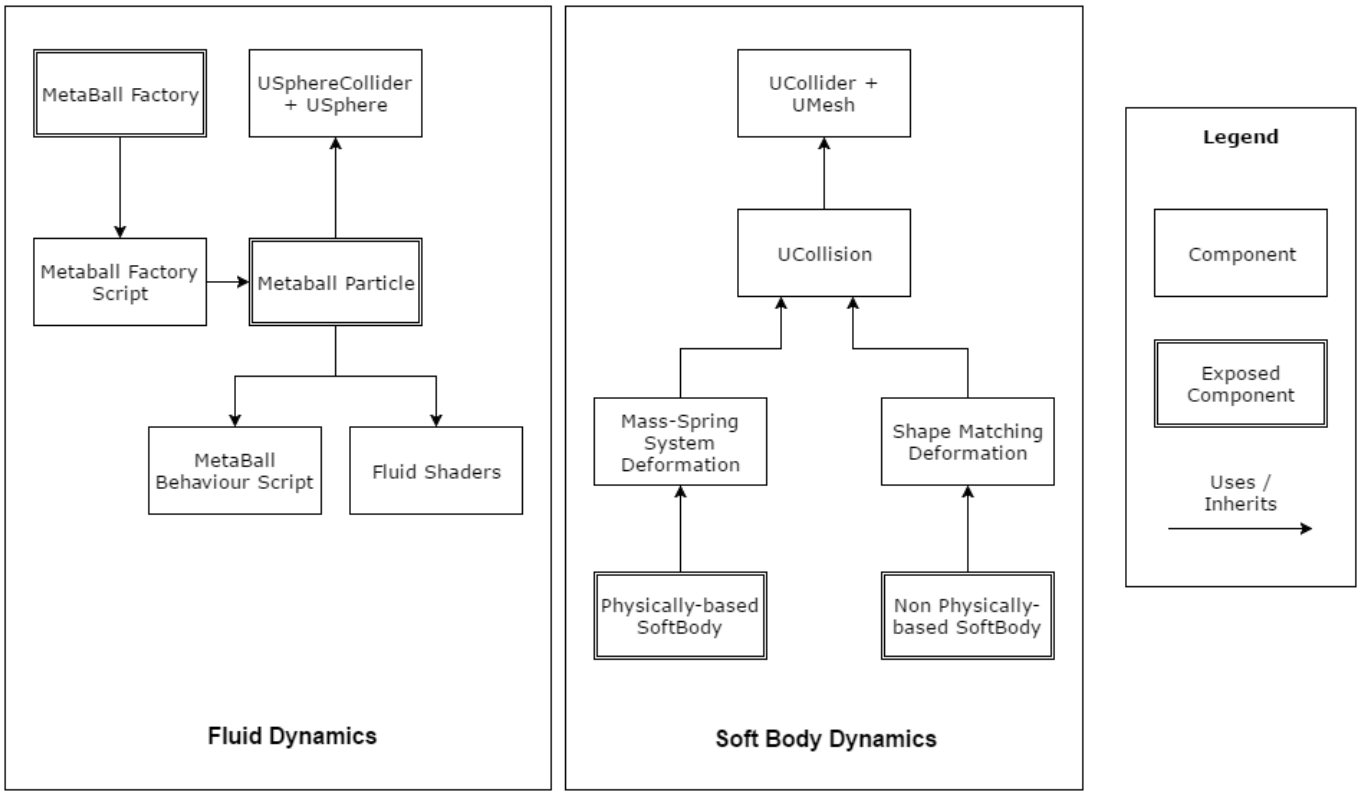
The production of the soft body physics component via shape matching should aid the development of video games that employ deformable solids. The mass-spring system component will be developed to try to include some of its accuracy measures into the shape matching method, or see if it can render in real-time fast enough to be used in video games.

Compare and contrast the two simulation methods thoroughly and rate their performance, accuracy, response and stability with respect to each other.

The open-source nature of the plugin will enable development to continue regardless of whether or not the initial developers contribute, allowing a community of developers to fork and build upon this project.

RESEARCH METHODOLOGY

Architecture Diagram



The overall architecture details that the system is essentially split to two major components. Despite the project being open source, and the code being provided with the plugin, some components are marked ‘exposed’. These components will be the only ones developers will have to configure when using the plugin, but they may edit the code if they require additional or alternative functionality.

Soft body deformation

The game engine uses component based development to add features and functionality to game objects. The rigid body component is added to meshes to add physics actions and collision handling to the object.

A ‘soft-body’ component will be produced to be added to game objects. This would contain the script and functionalities to handle the soft body deformations for the object.

The script would then expose some properties such as elasticity to the editor. This would allow developers using the component to use the component without having to change the code to suit their requirements.

The deformation of meshes will primarily be achieved using two methods: Shape Matching and Finite Element Simulation. The components for these will be developed in parallel with the aim of comparing the two at completion and if possible, see if the two can be merged to form some form of hybrid that manages to balance the two out.

Shape Matching

The main idea behind shape matching is to model the object as a set of particles with initial positions and masses, and in such a way that the particles of a mesh are not connected to each other and do not interact with each other.

The particles do however respond to external forces and collisions. After the reaction to an external force, the particles’ positions change relative to their starting position, relative to the centre of mass of the object. For each time step, the particles are moved to the goal position, which is calculated using its original and actual positions.

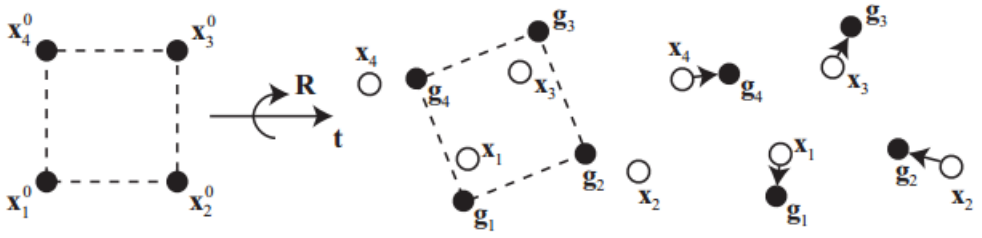


Figure 2 Shape Matching, by Müller et al. (2005)

X_i^0 is matched with x_i, the particles are then pulled towards their goal positions g_i.

The matrix form of the force on each vertex i, is given by the equation

$$\sum_i w_i(R(x_i^0 - t_o) + t - x_i)^2$$

x_i^o = original position
x_i = current position
R = rotational matrix
w_i = weight of the vertex
t_o = translation to the centre of mass of the original shape
t = translation to the centre of mass of the current shape

The translation vectors t and t_o can be calculated by the following equations

$$t = x_{cm} = \frac{\sum_i m_i x_i}{\sum_i m_i}$$

$$t_o = x_{cm}^o = \frac{\sum_i m_i x_i^o}{\sum_i m_i}$$

x_{cm} = shape's actual centre of mass
x_{cm}^o = centre of mass of the original shape
m_i = mass of the vertex = w_i.

Rotation is calculated with the positions of the particles relative to the centre of mass.

$$q_i = x_i^o - x_{cm}^o$$

$$p_i = x_i - x_{cm}$$

The linear transformation matrix A, can be calculated and then broken through polar decomposition

$$A_{pq} = RS$$

The above equations have been used to obtain a solution for R and S, which in turn were used to calculate the goal positions

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

Solutions for the position and the velocity for each vertex is then given as follows

$$x_i(t+h) = x(t) + hv_i(t+h)$$

$$v_i(t+h) = v(t) + \alpha\frac{g_i(t) - x_i(t))}{h} + h\frac{f_{ext}(t)}{m_i}$$

Figure 3 Equations from Muller et al. (2005)

Spring Mass Model

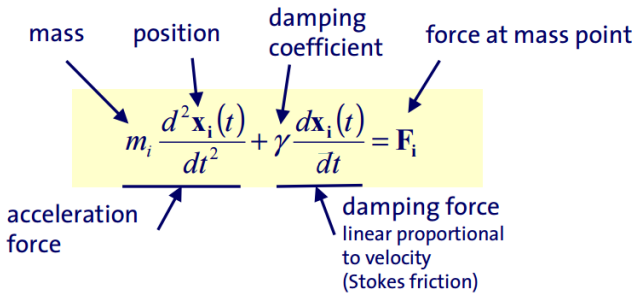
The mass-spring system works by modelling each particle or vertex as a point mass and the connections between particles as elastic springs. Dynamics are then computed with these assumptions.

Hooke's law states

$$F = k\ (\ L - l\ )$$

Where k is the spring constant, L is the initial spring length and l is the current spring length For each point the net force acting on it is equal to the sum of all the spring forces acting on it and the external forces that act upon it.

The motion equation for a mass m at a time t can be given as



Verlet integration gives us the following equations.

$$\mathbf{x}(t+h) = 2\mathbf{x}(t) + \mathbf{x}(t-h) + h^2\mathbf{a}(t) + O(h^4)$$

$$v(t) = \frac{\mathbf{x}(t+h) - \mathbf{x}(t-h)}{2h} + O(h^2)$$

Figure 4 Equations from Teschner et al. (2003)

These equations will be used over Newton's equation in Langrange form because of the stability it offers.

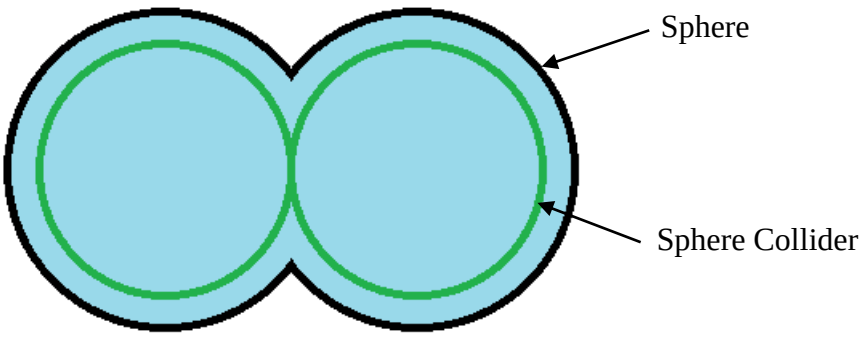
Fluids

Behaviour

Existing fluid simulation applications mostly use particle based method, which uses complicated algorithms and procedures to model fluids. A fluid will be represented by a set of particles, sometimes several thousand, which all operate under the algorithms individually. Particle based methods are exceptionally accurate and realistic. The trade-off is that this method is unbelievably expensive. Some renders with several thousand particles can easily take up to an hour.

The grid based method offers a fair balance between performance and realism. The fluid is represented as a grid with particles being represented as a single cell in the grid. Increasing the resolution of the grid can improve visuals at the expense of render time. In practice the system is not used often because the performance gain over particle based models is not worth the poor interactions.

The game engine does not provide any built-in features to model fluids. In the proposed plugin, fluid simulation is implemented using the concept of metaballs. Here, an approximation to metaballs is implemented through spheres in the game engine and sphere colliders. The engine's sphere colliders which typically bound the volume of spheres, are given a size slightly smaller than the, actual sphere. Hence, when two spheres get closer to each other, they will give an illusion of merging with each other due to their smaller colliders, which are actually touching each other. Through this, interaction between fluid particles is created.



The metaball particle is created as a prefab and a particle script that decides the behaviours of the metaball model will be attached to the particle prefab. Properties such as mortality, gravity, and flow speed are exposed through public variables, through which developers can change the values as desired.

Particle behaviour will be modelled after the Navier-Stokes equations. Simple surface tension may be modelled using flocking algorithms. Flocking describes the behaviour of particles as if they were in a flock, where particles would steer themselves based on rules of separation, alignment and cohesion.

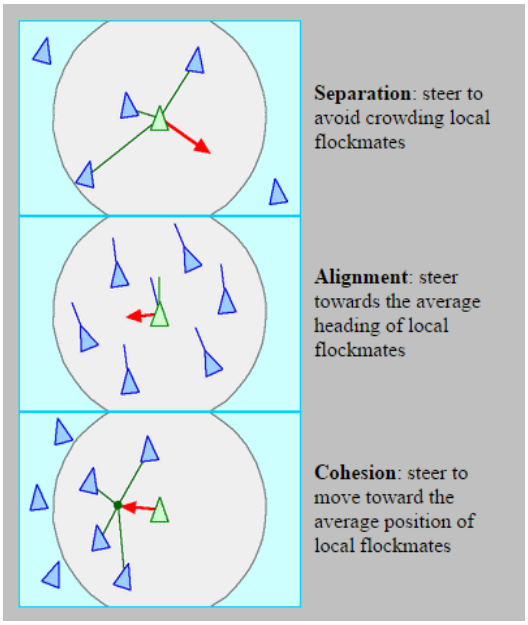


Figure 5 From Laster et al. (2011)

Metaball factory and shaders

The particle factory script creates metaball objects from the prefab. This script will be able to configure the spawned metaballs emitted. Settings include rate of emission, direction of emission, if the emissions should be in bursts, the lifetime of the metaballs and so on.

While metaball approximation is used to feign fluids, special shaders are programmed to resemble varying types of fluids, and improve the visualization of metaballs by hiding the overlapping edges and other vertices, as well as adding visual effects like transparency. Vertex and fragment shaders will be scripted to handle the refraction in fluids like water.

Other simple implementations of fluids using metaballs use only textures to fabricate water. This approach though works, requires several textures for different types of water and would only work for water (if the textures are for water). The texture method for shading metaballs requires a texture for every possible type of fluid the plugin would try to represent, which is not an approachable method of shading.

Writing our own shaders allows us to circumvent this problem, albeit giving up extremely realistic visuals for scalability. This also gives us the flexibility of rendering the metaballs the way we see fit, dealing with highlights, refractions and so on.

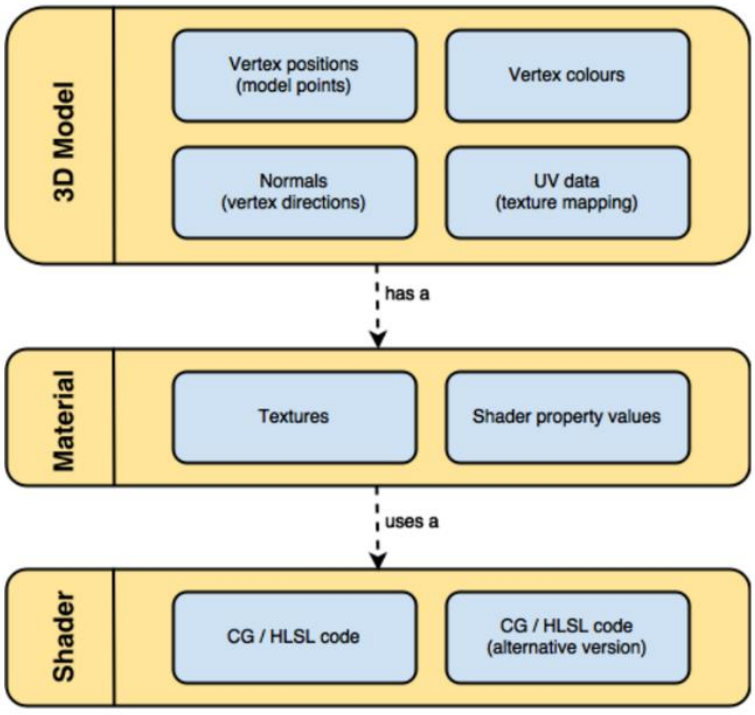
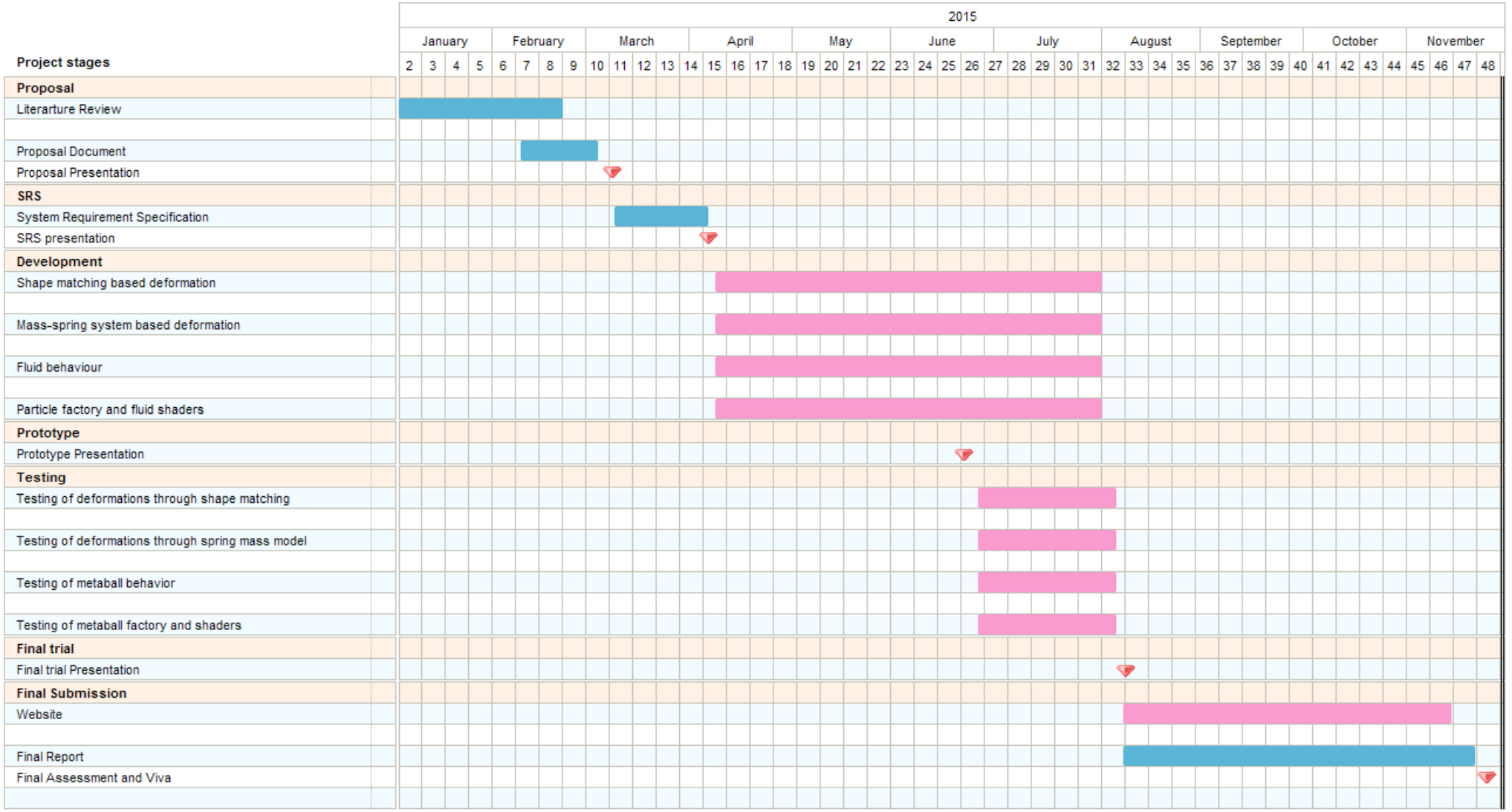


Figure 6 An Introduction to shaders – A. Zucconi (2015)

Project Gantt Chart



DESCRIPTION OF PERSONNEL AND FACILITIES

Member	Component	Task
M.Y. Alimudeen	Mesh deformation using shape matching	• Understand mathematical model behind shape matching • Develop matrix calculation component • Develop shape matching component • Test component with convex and concave meshes, with increasing polygon counts • Optimise component for 30, 60 and then 120 fps • Combine component with elements from mass-spring system
E.M.D.C.M. Ekanayake	Fluid shaders and particle generator	• Develop particle generation component • Develop lit shaders for basic liquids • Develop unlit shader • Understand mathematical model for refraction • Develop refraction shaders
S.M.D.R. Kithsiri	Mesh deformation using spring mass model	• Understand mathematical model behind mass-spring model • Develop component to handle synchronise mesh deformation with collision deformation. • Develop mass-spring deformation component • Test component with convex and concave meshes, with increasing polygon counts

		• Optimise component for 30, 60 and then 120 fps • Combine component with elements from shape matching
W.M.G.N.P. Kumara	Fluid behaviour	• Create metaball particle • Develop basic particle functionality, like lifetime, and so on. • Develop particle interaction component • Develop simple flow and surface tension component using flocking • Understand mathematical model for Navier-Stokes equations • Develop advanced particle functionality

REFERENCES

[1] M. Müller, D. Charypar et al. “Particle-Based Fluid Simulation for Interactive Applications” - *Eurographics/SIGGRAPH*, San Diego, California, 2003.

[2] Y. Yu, H. Jung and H. Cho, "A new water droplet model using metaball in the gravitational field", Computers & Graphics, vol. 23, no. 2, pp. 213-222, 1999.

[3] R. Geiss, "Ryan's Guide to MetaBalls (aka Blobs)", *Geisswerks.com*, 2016. [Online]. Available: http://www.geisswerks.com/ryan/BLOBS/blobs.html. [Accessed: 14- Jan- 2016].

[4] S. Laster and K. Bailey, "Flocking Algorithms", Flocking Fish, 2011. [Online]. Available: https://flockproject.wordpress.com/2011/05/14/flocking-algorithms-2/. [Accessed: 07- Mar- 2016].

[5] Gourlay, Michael J. "Fluid Simulation For Video Games (Part 1) | IntelÂ® Developer Zone*". Software.intel.com*. N.p., 2012. Web. 1 Mar. 2016.

[6] K. Crane, I. Llamas, S. Tariq and H. Nguyen, *GPU Gems 3*. Boston, MA: Addison-Wesley Professional, 2007, pp. 633-675.

[7] J. Groff, "An intro to modern OpenGL. Chapter 2.2: Shaders", *Duriansoftware.com*, 2010. [Online]. Available: http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-2.2:-Shaders.html. [Accessed: 28- Feb- 2016].

[8] W. Engel, *Direct3D ShaderX*. Plano, Tex.: Wordware Pub., 2002.

[9] Terzopoulos, Demetri et al. "Elastically Deformable Models". *Siggraph.* Pasadena: N.p., 1987. Print.

[10] Terzopoulos Demetri, and Kurt Fleischer. "Modeling Inelastic Deformation". *Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88* (1988): n. pag. Web. 13 Feb. 2016.

[11] "Havok Destruction". *Havok*. N.p., 2016. Web. 6 Feb. 2016.

[12] Muller, Matthias et al. "Meshless Deformations Based On Shape Matching". *TOG* 24.3 (2005): 471. Web. 8 Feb. 2016.

[13] Parker, Eric G., and James F. O'Brien. "Real-Time Deformation And Fracture In A Game Environment". *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA '09* (2009): n. pag. Web. 12 Feb. 2016.

[14] Teschner, Matthias. "Modeling Dynamic Deformation With Mass Points And Springs". 2003. Presentation.

[15] Liu, Tiantian et al. "Fast Simulation Of Mass-Spring Systems". TOG 32.6 (2013): 1-7. Web. 7 Mar. 2016.

[16] A. Rivers and D. James. “Fast Lattice Shape Matching for Robust Real-Time Deformation” - *SIGGRAPH*, 2007.

[17] Sharma H. *Soft Body Deformation Dynamics Based On Shape Matching*. 1st ed. Dorset: N.p., 2013. Print.

[18] M. Desbrun, P. Schröder and A. Barr, in Interactive animation of structured deformable objects - *Proceedings of Graphics Interface*, Kingston Ontario, 1999.