Real-Time Deformation using a Modified Finite Element Method

MSc Computer Animation and Visual Effects
Bournemouth University
Zoe Sams
i7698049

August 2015

Abstract

Every day, we interact with deformable objects such as cloth, hair, paper, and more. Within computer graphics these deformable bodies need to be simulated, whether it is skin for a medical simulation, an application to test stress levels on a bridge for an engineer, or just simply to look believable within a virtual landscape. Methods such as the mass spring model have allowed video games to efficiently recreate the movement of 2D soft bodies such as cloth and hair, however the simulation of more solid objects using this method has issues.

In this project, a modified Finite Element Method has been used in order to simulate soft body deformations of thicker objects in a real-time environment, intended for use either as a tool to export these elements or to show the potential applications of the finite element method within video games. The application was created in C++, and uses OpenGL, NGL and Eigen libraries. A visually believable deformation is created within the application, which could be used as a tool for artists as well as a showcase of the methods in real time. Finally, the application has been running with an average framerate of 82 frames per second (fps), exceeding the aim of 60 fps.

Keywords: finite element method, matrix inversion, real-time deformation, video games

Acknowledgments

I would like to extend my thanks to so many people that have helped support me throughout the course of this project. Most notably, I would like to thank Jon Macey for his support and guidance throughout the year.

Contents

1 Introduction **1**

2 Previous Work **3**

2.1 Soft Body Simulation 3

2.1.1 Commonly Used Deformation Methods 3

2.1.2 Finite Element Method 4

3 Implementation **6**

3.1 Initial Set Up 6

3.2 Pre-processing 8

3.2.1 Matrix Computation 8

3.2.2 Boundary Nodes 9

3.3 Solutions 9

3.3.1 Node Displacement 9

3.4 Post-processing 10

3.4.1 Rendering 10

4 Results and Discussion **12**

5 Future Work **14**

6 Conclusions **16**

List of Figures

1 Remeshed cuboid with high detail in the z-axis 7

2 On the left, forces applied without boundary nodes (to all nodes). On the right, forces applied to specific nodes, while the rest are immovably fixed. 9

3 Low level of detail to show node, neighbour, and element deformation within a cube 10

4 Visually believable deformations of an object 12

5 NVIDIAs FleX Cloth Tearing Simulation, where there are still visible polygons 15

6 Original UML diagram 17

1 Introduction

Deformation and destruction are visible in many aspects of everyday life, as technically all objects in the real world are soft body objects. For many years now, soft body deformation has been a prime area of research within not only computer games, but the entire computer graphics community. Games, film, engineering, and medical industries all benefit from softbody simulations, and have invested many hours collectively into this area of research. With the ever increasing realism within video games and interactive graphical applications, these deformations are an important aspect to be recreated.

Simulation of all objects as deformable bodies is unrealistic within a real-time virtual environment. Instead, the main objects of focus are often thinner sheets such as plastic, metal, cork and wood. While the mass spring model is great for cloth, it lacks the ability to be used for more solid objects. The finite element method allows the recreation of solid objects through remeshing the model into smaller elements.

The aim of this project was to find a method of object deformation which is visually believable, while the application still runs at a minimum framerate of 60 frames per seconds (FPS). These two factors are arguably the most important within a video game environment, as a players immersion can easily be broken if the frame rate drops or a simulation does not behave expectedly. For instance, if a rigid body is thrown at a soft body, the user will expect the soft body to deform at the point of impact. Methods of deformation and fracture simulation, such as prerendered or prescored objects (Fedkiw et al., 2009), will mean that the object already has a set area where it will deform, and the deformation will be repeated every time the user interacts with that object. For this reason, a physically based dynamic animation should be created in order to create a believable simulation. For implementation more specifically, this application was to be created using C++ and OpenGL, and demonstrate a deforming object in a real-time environment. The application would be able to save and load inverted stiffness matrices which could be used in further calculations in a video game setting, and also save out node positions to be loaded into software packages such as Houdini for rendering.

Within this paper, other viable methods of deformation are investigated and compared in order to accurately convey why the finite element method was specifically chosen for this project. Then, previous research into the finite ele-

ment method is presented. The implementation within this project is outlined, and the results of the application are compared with the initial aims of the project. Finally, additions and further areas of research which could expand the project in the future are suggested.

2 Previous Work

2.1 Soft Body Simulation

2.1.1 Commonly Used Deformation Methods

Every object and material within the real world will deform under a certain amount of pressue, however as Millington (2010) discusses, all objects have a different resistance to being deformed. This means that, technically, every object is a soft body. While it is implausable to calculate every object within a virtual environment as a deformable object, there have been many investigations into deformation techniques and techniques for soft body simulation.

The mass-spring model is commonly used within many real-time computer graphics applications to create soft body simulations, especially for representation of cloth and hair. Feng et al. (2006) discuss the usage of the mass-spring system for garments, describing implementation as a grid of nodes or particles interlinked with a system of springs. While this method is efficient, using simple stress calculations, it has its disadvantages also. One of the main issues with the mass-spring model is the "super-elastic" effect (Provot, 1995), in which the method starts to overstretch around boundary nodes. This creates unrealistic deformations, and means creating any kind soft body with a solid object with this method is far more difficult. Not only this, but the way in which the mass-spring model is constructed means that it struggles with particularly stiff springs (Nissen, 2014). This makes the use of the mass-spring model unsuitable for rigid or stiff simulations, as springs are notoriously unstable.

Another method of deformation commonly used is Free-Form Deformation. A lattice is created around an object made up of B-Splines. Each area within the lattice controls an area of the objects mesh and, when the control points of the spline are moved, the mesh is deformed by this area (Sederberg et al., 1986). This method is great for choreographed deformations, such as a tool for artist, where the user knows exactly how they want the object to deform. Within an interactive environment however, choreographed deformations are not as believable, as points of impact or fracture cannot be accurately predicted.

2.1.2 Finite Element Method

The Finite Element Method (FEM) is a way of predicting some unknown quantity such as partial differential equations (Henwood et al, 1996). FEM is industry standard for scientific research. Ross (2012) discusses the importance of the finite element method in engineering being used to determine weaknesses in structures such as bridges. Within these applications, mathematical accuracy is key, however in video games and visual effects we can afford to cut corners as long as the final output is visually believable. Though the use of FEM is less popular within interactive applications, a modified method can be implemented in real-time.

The main idea behind FEM is to replace a complicated shape with a mesh of smaller shapes called elements. These elements are unique to each new problem, and therefore must be designed with that in mind. A higher number of elements means more accurate results, however the drawback to this is that more calculations are needed, causing a higher computational expense. Compromises must be made in order to create an aesthetically pleasing simulation as well as an efficient one.

An important feature of the finite element method, is that it can be expanded to include fracture simulation as well as deformation. OBrien has written two papers on both ductile and brittle fractures. The initial paper on brittle fracture (1999) outlines the finite element method base to be used, with elements taken from elastic theory. Once the deformations of the objects have been applied, the internal stress forces at each node are calculated. After exceeding a certain threshold, these nodes are split in two, and a fracture plane is created to determine the direction of the fracture propagation. Extending from this, OBrien (2002) discusses that to create ductile versions of these fractures, plasticity must be added into the original base method from 1999 by redefining the strain method to elastic strain.

Focusing specifically on games, OBrien and Parker (2009) presented a paper discussing the use of the Finite Element Method in real-time. In order to efficiently compute certain algorithms, OBrien proposes using parallelism on certain parts of the algorithm. As this paper points out, an important feature of the finite element method and the use in video games is the ability of the soft body to interact with a rigid body. Most objects within games are rigid bodies, so the ability of one to interact with the other in the form of collisions is crucial to the believability of the simulation within a game environment.

Another area that O'Brien and Parker point out as important is the visual quality of the final piece, referring to the frame rate and the believability of the simulation. If there is any drop in frame rate at any point in the game, the immersive experience will be lost.

Rebours (2001) discusses his implementation in real-time of the finite element method. Although not used specifically within video games, this method involves the inversion of a matrix instead of full computation. This means that instead of using complex equations to derive the solution for $F = KU$ where F is the force vector applied, U is the displacement of the nodes and K is the stiffness matrix of the nodes, one matrix inversion can be calculated, stored, and loaded during the loading of a game level. Due to the assumed efficiency and simplicity of this method, it was chosen to be implemented.

3 Implementation

The finite element method can be constructed with many different shapes, which allows a great deal of flexibility when creating a mesh of elements. This also means that appropriate element shapes should be chosen depending on the application. Extremely thin sheets such as cloth, paper, or hair, have no need for a solid interior structure as it would barely be seen. In these situations, a 2D element such as a triangle or square are perfect, and additionally a triangle means an easy to render mesh. Less nodes also mean less complex matrix calculations further down the line. 2D elements are not suitable for every implementation however, as even when still focussing on thin sheets, those with more structure, such as cork or metal, need 3D elements in order to accurately display the inside structure of the material as well as the outer shell.

Creating cuboid elements allows the addition of this substance. As it is a convex shape, it also allows fast and accurate collision detection to easily be included within the application.
There are a few main steps within this modified finite element method, which can be split into:

Initial Set Up
Material and geometry properties
Remeshing/subdividing

Pre-processing
Computation of each stiffness matrix
Boundary conditions

Solutions
Application of the boundary conditions
Resolution of $F = KU$

Post-processing
Rendering

3.1 Initial Set Up

A fundamental aspect behind the finite element method is taking a complicated shape and breaking it down into a network of simple elements that are approximately equivalent to the original shape. To do this, an object must

be remeshed. This new mesh is unique to each case, and therefore it can be often seen as more appropriate for a element mesh to be created in an external package such as Maya. In this implementation, a cuboid shape will be broken down into a series of smaller cuboids. Simplicity is an important element within video games. The simpler we can make the problem, the quicker the solution can be processed. By creating cuboid elements, we can do simple remeshing calculations for objects which are most likely to be deformed or destroyed within video games such as walls, planks and beams. Cubes are also simple convex shapes, meaning collision detection is far easier than with an element of a complex shape.

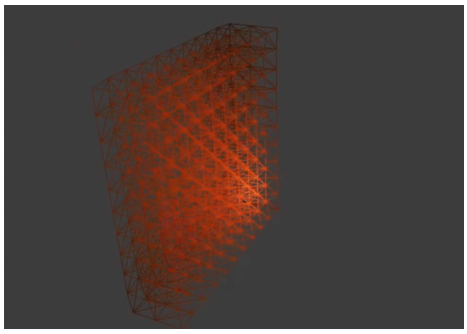

Figure 1: Remeshed cuboid with high detail in the z-axis

A function takes the current dimensions of the object and the mesh detail that the user inputs into the program, and divides the length by the level of detail intended. From there, the function loops through, adding a new element at each iteration. Each element within the new mesh will share nodes with other elements, so when an element is created it checks the position for a node to be added against a vector of nodes already created within the mesh titled m_globalNodes. If there is already a node in that position, that node is pushed back into an array specifically to be used in that element. Along with this, a material is created and assigned to each element. This material takes the elastic modulus and Poisson's ratio values input into the application and assigns them to be used within the next step of matrix computation.

3.2 Pre-processing

3.2.1 Matrix Computation

In order to determine how the network of nodes interact with each other, a stiffness matrix must be created. Each element within the finite element method has a stiffness matrix, and these matrices must be combined to form one global stiffness matrix. The construction of the global stiffness matrix is fairly simple to comprehend as each element will share some nodes with another element. Cube elements are constructed out of 8 nodes, and in a 3D scenario each node has 3 degrees of freedom, meaning that the stiffness matrix for each element in this application is a 24x24 matrix.

Within the program, it was decided that an external library and reference would be used for the construction of the stiffness matrix. The algorithms executed within the CalculateStiffnessMatrix functions of each element are based off of those presented by Rebours (2001), which were originally generated by Maple. The elastic modulus, Poisson's ratio, and dimensions of each element are combined in order to form connections between each of the nodes which dictate how they are to move in relation to each other.

Another aspect of matrix computation within this program is the inversion of the stiffness matrix. With 24x24 elements, the inversion of this matrix is arguably one of the most computationally expensive procedures within the application, and it is therefore handled by the Eigen linear algebra library. Within the library, there are functions within the MatrixXd class which invert a matrix greater than 4x4, which are used to invert the stiffness matrix of each element. This matrix is then saved out to a comma separating value file, so that it can be used again rather than recomputed. Not only does saving and loading of matrices make the current application faster, but as a game environment is often designed, the entirity of the pre-processing and initial set up steps could be done within an external application and loaded directly into the game. Here, the only steps needed would be $U = K^{-1}F$ and the application of boundary nodes.

3.2.2 Boundary Nodes

To complete the set-up of the finite element mesh, boundary conditions must be set to determine which areas of the simulation are immovably fixed. These constrained points could be areas such as the floor or ceiling, as if to simulate something pinned to a wall or extremely heavy so as not to be moved from the ground. Implementing the boundary nodes within an application is simple, as we zero out the values and remove them completely from the stiffness matrix. This, however, means that once the matrix is inverted, the boundary nodes cannot be changed.

Instead, the implemented method zeros out the forces to be applied to those nodes rather than the stiffness matrix. This means that even after the matrix has been inverted, the boundary nodes can be changed by simply changing which nodes have a zero force applied to them.

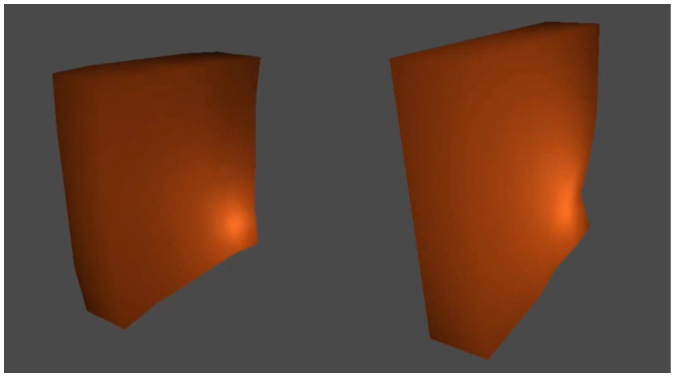

Figure 2: On the left, forces applied without boundary nodes (to all nodes). On the right, forces applied to specific nodes, while the rest are immovably fixed.

3.3 Solutions

3.3.1 Node Displacement

As described before, the main equation aiming to be solved through this set up is $F = KU$ where F is the force vector applied, U is the displacement of the nodes and K is the stiffness matrix of the nodes. In order to calculate the displacement of the node, many implementations of FEM use Gaussian

elimination. Rao (2005) explains that the Gaussian Elimination method is a rather simple concept, in which you have multiple unknowns in multiple equations, and can combine these equations in any way in order to eliminate some of the unknowns until your answer has been found. This is a very iterative process however, and can take quite some time. In order to get as efficient an implementation as possible, we instead use a matrix inversion.
Using the inverse functions from the Eigen library, the stiffness matrix inverse is calculated, and the equation $U = K^{-1}F$ can be used instead. The displacement of the nodes can now be calculated simply by multiplying the inverse stiffness matrix and the force vector, making for a much simpler calculation. This application will show the resolution of $U = K^{-1}F$, however it can also save the inverse matrices out to comma separated value files, meaning a separate game engine could be used to handle the $U = K^{-1}F$ function. After the solutions have been derived, the last important step of the implementation is the rendering.

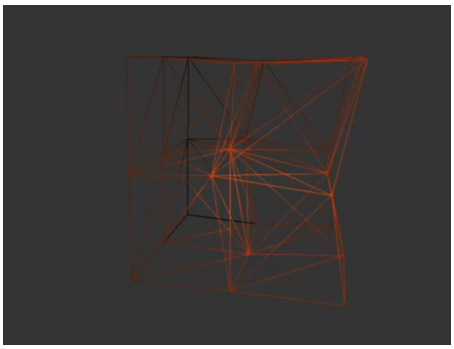

Figure 3: Low level of detail to show node, neighbour, and element deformation within a cube

3.4 Post-processing

3.4.1 Rendering

To visualise the elements within the mesh, OpenGL and NGL were used. Each element handled their own draw functions, which involved simply passing in the vector of node positions into 3D position vectors. As each element was constructed in the same way, the node order was also the same. The main

improvement which could have been made in this area is the rendering of inside faces. Elements check to ensure they do not create additional nodes when being constructed, however there is no method to check the redrawing of faces. Not only this, but the inside faces are rarely seen until a fracture has occured, so the program could run far more efficiently if faces that were not currently visible to the user were not rendered. Much like the stiffness matrices, the point positions can be exported by checking the "Save Frames" box within the application. This will save out the current positions frame by frame until the box is once again unchecked. This can allow for higher quality renders of the simulation.

4 Results and Discussion

Many implementations of the finite element method, such as those for medical and engineering simulation, require a high degree of mathematical accuracy in order to accurately predict stresses and deformations. In video games however, there are three main points to focus on: efficiency, frame rate, and believability.

As well as the code being efficient, one of the main aims for implementation within video games is a constant frame-rate. While closely related to efficiency, the purpose of frame rate focus is to ensure that the user is constantly engaged within the experience. Any drop in frame rate will quickly take away from the immersion of the player. One of the aims of this project was to ensure the completed application ran at a minimum of 60fps. The average frame rate recorded from this application on one machine was around 82fps, which successfully exceeds the outlined goal. The only areas in which a dip in frame rate were recorded were those including timer events (such as updating the camera) and those which involved matrix recalculations. Many video games aim for a frame rate of around 30fps, but with virtual reality becoming every popular, the need for high frame rates increases.

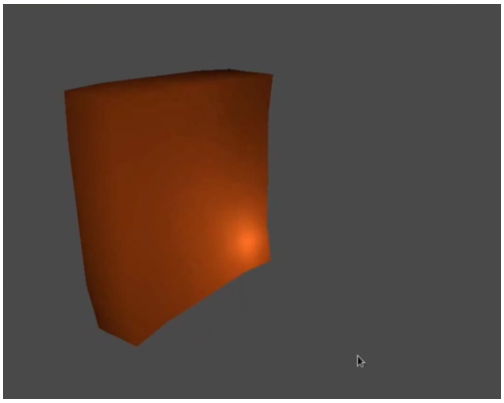

Figure 4: Visually believable deformations of an object

The most important factor in video games, however, is the believability. Scientifically, the simulation could be relatively inaccurate, but if the simulation is visually accurate enough for the player to believe that the deformation see before them is what could happen, then the simulation has succeeded. It is, obviously, unfavourable for the simulation to be incorrect, and the more

powerful games consoles and PCs become, the closer we can get to real-time mathematical accuracy.

The simulation created for this application is, on the whole, believable. Figure 3 shows the application with a low number of element. Here, there are clearly issues where polygons can be seen in the deformation. This issues can be resolved by increasing the number of elements within the mesh, seen in Figure 1, offering a higher detail mesh. Though this is slightly more expensive when running in real-time, the method implemented within the application means that most of the expensive calculations are done at load time, or when recalculating the mesh.

5 Future Work

There are additions that, were this project taken further in the future, could improve the application. The matrix inversion was added to simplify the calculations involved in updating the position of the nodes, and avoid iterative solutions and Gaussian Elimination techniques. The addition of multithreading would be a great addition to speed up the application. The most expensive calculation within the application created in this project is the inversion of a matrix. Multithreading this process would hugely benefit the computational speed of the program, as would transferring some calculations to the GPU. In contrast to this, a multithreaded application could also handle the complex calculations used to solve F=KU. Although this process would be slightly slower than a matrix inversion, it would give more accurate results, possibly still in real-time and hitting the 60FPS framerate aim set out in this project. Not only this, but it would avoid situations where matrices are unable to be inverted. This would allow all sorts of element shapes and material combinations.

The current application could also be extended to include fracture simulation. As discussed previously, OBriens method of creating fractures (2002) by measuring the internal stress is a great way of creating these splits. For use within this application, a new node could be created when the stress reaches above a certain threshold, and a fracture plane could be created. This fracture plane dictates the direction of the fracture propogation, and which nodes are now weaker and more likely to break. Combining this with more efficient rendering in the form of only rendering when faces are visible to the user (which are often attached to nodes with less than 3 neighbours) would make an interesting addition to the project.

Adaptive remeshing of an object could also be a great addition to this project. Within many fracture simulations, the main flaw is visible polygons. Figure 5 shows the cloth tearing simulation within NVIDIAs FleX, their particle based dynamics application (Macklin et al., 2014). While this program is one of the most efficient within current applications of real-time soft body tearing, there are still visible artefacts and polygons within the simulation. To get to a closer level of visual realism, adaptive remeshing must be investigated. Pfaff et al. (2014) discuss the use of the adaptive remeshing library ARCSim along with their algorithms for cracking and tearing of this sheets. The visual results of this paper are very aesthetically pleasing, and aiming for this in real-time

should be a future goal. However, the paper discussed is still currently running rendered simulations, and it may be some time before we see similar visual results in video games. Interactive applications are, however, getting closer to this level. Adaptive meshing and tessellation in real-time is a large area of focus right now, with Call Of Duty: Ghosts (2013) using adaptive mesh tesselation for sniper rifles when they are brought closer to the player camera.

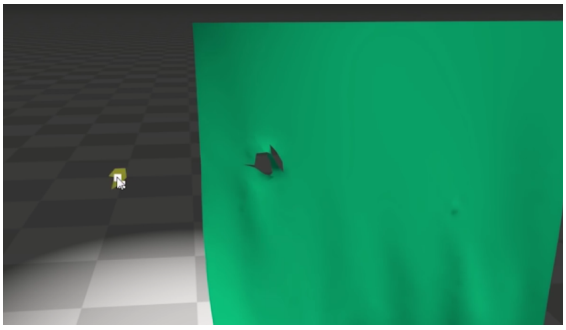

Figure 5: NVIDIAs FleX Cloth Tearing Simulation, where there are still visible polygons

6 Conclusions

While the techniques implemented within the final application have their issues, overall the project was successful. A believable soft body deformation is created and runs above the target speed of 60 FPS. Although this application was created to show that a finite element based solution could be used in a real-time environment such as a video game, if it were to be expanded slightly its best implementation/usage would be as a tool for artist to use. An artist could create the material with the characteristics they wanted, remesh the desired object, test force applications on certain areas, and export the matrix inverse for calculations within the final game environment.

One of the main flaws of the technique implemented within this project is the limitability. As discussed, updating the elements shape and material properties interactively means that matrices can become singular, and in turn unable to be inverted. This issue can be avoided as it is unlikely that a matrix will need to be updated within a game-environment, however it still should not be ignored. Within a tool based environment, the prompt discussing that the matrix has become singular is a temporary solution, while a multithreaded application that could handle the complex gaussian equations needed to properly calculate the final results could avoid matrix inversions entirely. Video games are getting closer and closer to becoming interactive films with high levels of detail and believability. Dynamically simulated materials are an important part of creating this immersive digital environment, and are quickly becoming a necessary part of gaming.

Appendix A

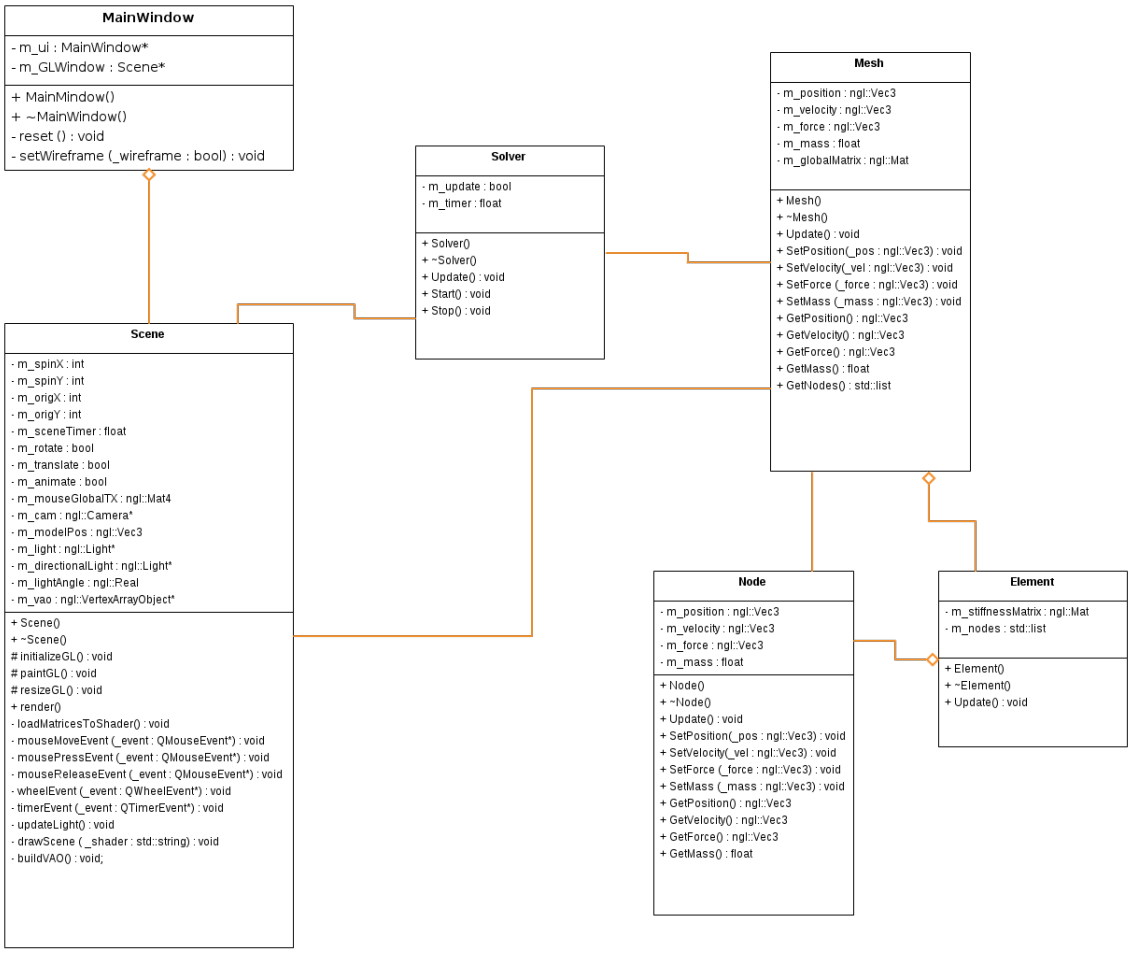

Figure 6: Original UML diagram

References

Call of Duty: Ghosts. 2013. [disk] Xbox One. Activision.

Fedkiw, R., Schroeder, C., Su, J. 2009. Energy stability and fracture for frame rate rigid body simulations. *In: Proceedings of ACM SIGGRAPH,* 2009. pp 1-10.

Henwood, D., Bonet, J. 1996. *Finite elements: a gentle introduction.* London: Macmillan Press.

Macey, J. 2013. *SimpleNGL* [online]. Available from: https://github.com/NCCA/SimpleNGL [Accessed June 2015]

Macklin, M., Muller, M., Chentanez, N., Kim, T.Y. 2014. Unified particle physics for real-time applications. *In: Proceedings of ACM SIGGRAPH,* 2014. New York: ACM Press.

Millington, I. 2010. *Game physics engine development: how to build a robust commercial-grade physics engine for your game.* FL: Taylor & Francis Group. pp 336.

Nissen, M.S. 2014. *Towards a simpler, stiffer, and more stable spring* [online]. Available from: http://www.gamedev.net/page/resources/_/technical/math-and-physics/towards-a-simpler-stiffer-and-more-stable-spring-r3227 [Accessed June 2015]

O'Brien, J.F., Hodgins, J.K. 1999. Graphical modeling and animation of brittle fracture. *In: Proceedings of ACM SIGGRAPH,* 1999. New York: ACM Press/Addison-Wesley Publishing Co., pp 137-146

O'Brien, J.F., Hodgins, J.K., Bargteil, A. W. 2002. Graphical modeling and animation of ductile fracture. *In: Proceedings of ACM SIGGRAPH,* 2002. San Antonio, TX: ACM Press, pp 291-294

O'Brien, J.F., Parker, E.G. 2009. Real-time deformation and fracture in a game environment. *In: Proceedings of ACM SIGGRAPH,* 2009. New Orleans: ACM Press, pp 156-166

Pfaff, T., Narain, R., de Joya, J.M., O'Brien, J.F. 2014. Adaptive tearing and cracking of thin sheets. *In: Proceedings of ACM SIGGRAPH,* 2014. Vancouver: ACM Press, pp 1-9

Rao, S. 2005. *The finite element method in engineering* Oxford: Elsevier Inc.

Rebours, P. 2001. *Real-time deformation of solids, Part 1* [online]. Available from: http://pierrerebours.com/blog/real-time-deformation-solids-part-1 [Accessed July 2015]

Sederberg, T.W., Parry, S.R. 1986. Free form deformation of solid geometric models. *In: ACM SIGGRAPH computer graphics*. 20(4). pp 151-160.