FRAMEWORK FOR PHYSICALLY BASED SHATTERING

MAYA API PLUG-IN

MASTERS THESIS

HANNES L. RICKLEFS

N.C.C.A BOURNEMOUTH UNIVERSITY

5th September 2005

Contents

1 Introduction **1**

2 Previous Work **2**
2.1 Introduction . . . 2
2.2 Animating fracture . . . 2
2.3 Mechanical simulations . . . 3

3 Technical background **4**
3.1 Maya programming interfaces . . . 4
3.1.1 MEL . . . 4
3.1.2 C++ . . . 4
3.1.3 MEL or C++ . . . 4
3.1.4 Maya approach . . . 5
3.2 Algorithms . . . 6
3.2.1 Lattice Model . . . 6
3.2.2 Fracture Mechanics . . . 7
3.2.3 Other Algorithms . . . 7
3.2.4 Class Design . . . 7
3.2.5 Development Environment . . . 8

4 Implementation **9**
4.1 Introduction . . . 9
4.2 Internal structure . . . 9
4.2.1 Lattice creation . . . 10
4.3 Surface approximation . . . 12
4.4 Shattering . . . 14
4.4.1 Shattering design . . . 14
4.4.2 Random shattering . . . 15
4.4.3 Force shattering . . . 15
4.4.4 Force random shattering . . . 17
4.4.5 Force radius shatter . . . 17
4.5 Polygonisation . . . 18
4.6 Computation of the plug-in . . . 19
4.7 Problems . . . 21
4.7.1 Initial approach . . . 22
4.7.2 Maya API . . . 22

5 Conclusions and future work **25**

A Class Diagram **26**

B Email - Contacts **28**

C Email - Alias support **31**

D Code - Examples **34**
D.1 Lattice Points 34
D.2 Evaluating ShatterPieces 34
D.3 Evaluate ShatterPiece according to neighbour connectivity 35

List of Tables

3.1 MEL or C++ comparison 5

List of Figures

3.1 Dependency Graph after shatterForce command. 5

3.2 Illustration of a lattice model for wood. 6

3.3 Voxel adjacency's in 3D discrete space: (1-faces) the six voxels that are 6-adjacent to the voxel at the center, (2-edges) the eighteen voxels that are 18-adjacent to the voxel at the center, (3-vertices's) the twenty six voxels that are 26-adjacent to the voxel at the center [KCY93]. 7

4.1 InternalCell creation. 11

4.2 Object filled with InternalCells of CellType inside. 12

4.3 Old versus New approximation method 13

4.4 Sequence diagram of shatter events. 14

4.5 Randomly shattered cube. 15

4.6 ForceLocator 16

4.7 Crack generated through force shatter 17

4.8 Glass wall shattered with force shatter radius. 18

4.9 Polysmooth operation 20

4.10 Initial cells structure creation with bounding boxes 22

4.11 MFnMesh::intersect problem 23

4.12 MFn::intersect problem with smaller bounding box. 23

Chapter 1

Introduction

This thesis outlines the masters project by Hannes Ricklefs as part of the MSc Computer Animation at Bournemouth University in 2005. The masters project aimed to develop a framework for physically based shattering as a C++ plug-in for Alias Maya [May05]. This thesis is divided into five chapters. The first chapter introduces the reader to the project. The second chapter provides a brief insight into previous work both in relation to fracture for computer graphics and mechanical simulations. The third chapter presents the technical background behind this project including an explanation of the Maya approach taken and other algorithms that were included or adapted for this implementation. The fourth chapter explains the actual implementation, outlining each of the components of the plug-in in detail, as well as the computation of the plug-in as a whole. The chapter concludes by listing some of the major problems encountered during the development of this master project. Finally, the fifth chapter gives concluding remarks and states future work that could improve this project.

The Appendices include a short introduction to email conversation with contacts in industry, email conversations with the Alias Support team and code extracts of sections referenced in the text.

All the source code for the project, as well as HTML and pdf documentation referred to the programs is included on the CD handed in with this masters thesis.

The terms object and mesh will be used interchangeably throughout this thesis and both refer to polygonal mesh data of Alias Maya [May05], or in Maya API terms the MFnMesh class.

Chapter 2

Previous Work

2.1 Introduction

Physics-based simulation has gained attention in many fields of computer graphics, including 3D game engines, computer animation for feature films and material science. Through the ever increasing demand for realism and detail, the complexity of such simulations has been steadily increasing, including many physical phenomena, from melting and flow[CMHT02], fire[NFJ02], smoke[FSJ01], explosions[FOA03], to fracture mechanics[OH99, PKA+05, OBH02, SWB01]. All of the aforementioned papers grave limitations in that they are either computational expensive or the application of the simulation is too specialised to be used within a production environment. In the initial stages of the project people in the industry were contacted and asked how they would approach physically based shattering of objects. They were further asked which features they would consider important when implementing a shattering solution. See AppendixB for an extract of the communications. All of them mentioned the distinct lack of good shattering tools and that none of them use the internal Maya shattering tool. The only viable commercial example can be viewed at http://www.blastcode.com. The underlying reason for the lack of such shattering tools is the difficulty of creating adaptable and controllable realism.

2.2 Animating fracture

The realistic animation of breaking objects becomes virtually impossible using any of the traditional computer animation techniques such as hand modeling and key-framing. The main problem arises from the fact that the breaking of an object usually creates many differently sized and shaped pieces. The complexity and number of these pieces makes modeling a time consuming and difficult approach. Simple methods, such as slicing the surface or the use of RenderMan shaders do not produce the distinctive look of shattered object pieces. Attempts to solve this problem were made as early as 1988 [TF88] when Terzopoulos and Fleischer presented a technique for modeling viscoelastic and plastic deformations. Even though their work was not intended to model breaking, it allowed the simulation of tearing cloth and paper. The most cited work in the area of fracture simulation is that of O'Brien *et al*. They provided methods for both, fracture of brittle, as well as ductile materials [OBH02, OH99]. Both approaches used continuum mechanics techniques developed in mechanical and civil engineering to model the behaviour of fracture, including crack initiation and propagation. Most recently, Mark Pauly *et al.* presented a new mesh-less animation framework for fracturing elastic and plastic materials. Central to there method is a highly dynamic surface and volume sampling method that supports arbitrary crack initiation, propagation, and termination, while avoiding many of the stability problems of traditional mesh-based techniques[PKA+05]. Unfortunately, this paper was published too late in order to incorporate the ideas into the implementation of the project.

2.3 Mechanical simulations

The main motivation for this masters project was to create a tool for fracturing wood. All of the aforementioned papers have one distinct disadvantage in that they only distinguish between brittle and ductile materials. The contacts in industry stated that it would be of benefit to them to have an application that could incorporate physical properties of different materials and not having to fake them through the use of textures.

Therefore, the initial intention was to implement the approach outlined in [Ric05] for shattering objects according to wood properties. The main field that does research into fracture and fatigue in wood is that of engineering. However, simulations need to be as accurate as possible in order to simulate/predict failure of wood in real world structures. Thus, these simulations cannot be included into a computer animation or visual effects production as these computations take several hours to compute and are in not controllable. Research in engineering concentrates on the modelling of wood micro structure [AHS96] and the damages of certain wood types as in [WKDL$^+$05]. The only book on fracture in wood is that by I. Smith *et al.* [SLG03]

Chapter 3

Technical background

This chapter outlines some of the technical aspects and algorithm used in the implementation of the masters project. In addition, it explains what context the masters project was developed for. It was clear from the outset that the application of this project should be as industry relevant as possible. As stated in section 2.2 , the majority of work in this field cannot be used within a feature film or games production. Therefore, the decision was made to implement the shattering framework as a plug-in for Alias Maya [May05]. Maya is the main 3D graphics package taught to the MSc Computer Animation and is used extensively throughout the industry. Given its extensive tool set, such as modeling, dynamics, animation, and rendering, it is used for all aspects of production work[Dav03].

3.1 Maya programming interfaces

3.1.1 MEL

MEL is an acronym for Maya Embedded Language. MEL is an interpreted language designed around a simple structure and syntax which makes it easier and more widely accessible than the C++ programming interface. MEL can be written, executed and debugged entirely within Maya. The main limitation of MEL compared with the C++ programming interface is its execution time. As a C++ program is complied it can execute actual native machine instructions. Whereas with MEL, Maya has to interpret and finally convert the instruction in to native machine code. Even though Maya is optimized in doing so it still lags far behind the corresponding C++ program. The main benefit of a MEL script in comparison to C++ is that there is no setup, complexity and compile cost of a C++ program [Dav03].

3.1.2 C++

Maya can be programmed using the standard C++ programming language. This is the most effective way of extending Maya. Through using C++ it becomes possible to create native Maya plug-ins that work seamlessly with the rest of the package. Programming is provided through the C++ API (application programming interface). This API provides a number of C++ class libraries that can be used to create a plug-in through either using or extending these classes [Dav03].

3.1.3 MEL or C++

Table 3.1 outlines the main advantages and disadvantages between MEL and the C++ API. It is up to the developer which of the two programming interfaces to use. It should be noted, however, that choosing one

does not automatically exclude the other. This project choose to use the C++ API to develop the relevant nodes and MEL for the connection setup between the nodes in order to provide the necessary computations.

	MEL	C++ API
Ease of use	+	-
Cross-Platform	+	-
Speed	-	+
Custom-Functionality	-	+

Table 3.1: MEL or C++ comparison

3.1.4 Maya approach

In order to explain the choices of method, this section gives a short introduction to Maya [1]. Generally any 3D application tries to use distinct modules where each module performs some computation and passes the resulting data to the next module. Hence, the functionality provided by separate modules can be encapsulated in a series of interconnected operators[Dav03]. Alias implemented Maya's core using this data flow paradigm [May05, Dav03]. This core is embodied in the Dependency Graph, DG. The DG is the heart of Maya, it provides all the fundamental building blocks that allow the creation of arbitrary data which is fed into a series of operations and result in some form of processed data at the other end. The data and their operations are encapsulated in the DG as nodes [Dav03]. Each node holds any number of slots that contain the data used by Maya. Nodes also contain an operator that can work on their data and produce other data as a result.

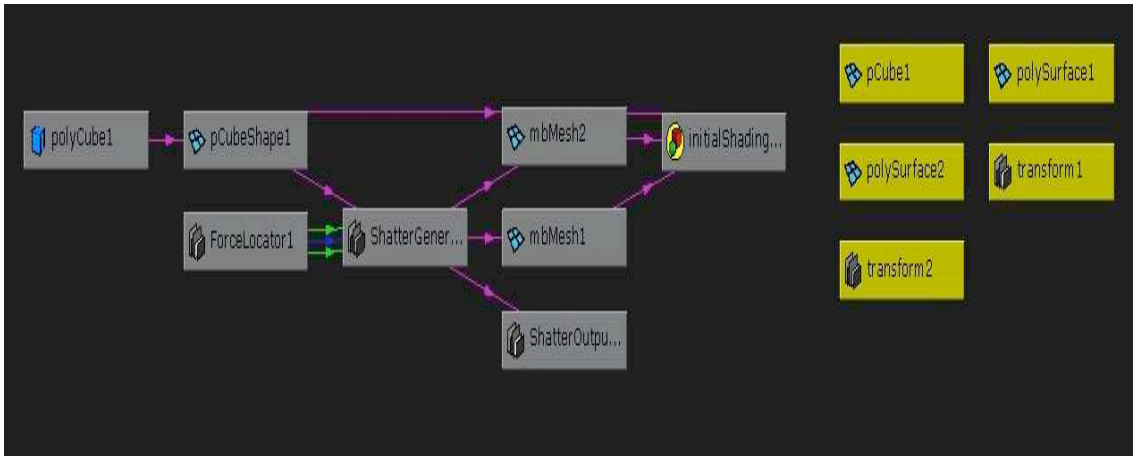


Figure 3.1: Dependency Graph after shatterForce command.

Figure 3.1 shows the resulting Dependency Graph after a polygonal cube has been selected and the created MEL command **shatterForce** has been executed. The custom nodes created by the Shatter plug-in: ForceLocator, ShatterGenerator and ShatterOutputLocator are shown and the relevant connections can be seen. This is a good example of how Maya's node based approach and the C++ API hides the complexity of the implementation and presents the user with simple nodes that can be connected.

Apart from the choice between MEL or C++ API, it had to decided whether the plug-in should consist of a single command or interconnection nodes. Given that the contacts from industry stated they would like to

[1] Please refer to the Maya documentation provided with each installation or consult [MAY05, Dav03] for a more in depth explanation.

have as much control over the actual shattering as possible the decision was made to use interconnecting nodes. The attributes of nodes are changeable and therefore allow the modification of the shatter after the command has been executed. To summarize, MEL was chosen for its command related functionality and the C++ API was chosen for any of the high computational functionality incorporated into nodes.

3.2 Algorithms

In general this masters project does not use and is not build around a specific algorithm. It rather takes ideas from existing algorithms and adapts them to apply to the application. As stated above the decision was made to apply this masters project to the fracturing of wood and to implement the approach used in the authors Personal Research[Ric05]. This approach is based around a Lattice data-structure and is the first stage of the implementation.

3.2.1 Lattice Model

With a lattice model, the traditional continuum representation of material is abandoned, and the material is represented by an array (or Lattice) of interconnected discrete bar elements. Properties of elements can correspond to physical micro-structural features. Lattice models have been used for a variety of heterogeneous materials [HR90], because disorder can be easily introduced into fracture and damage problems.

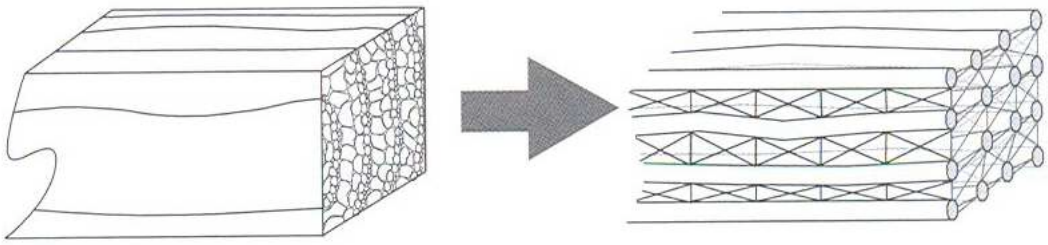


Figure 3.2: Illustration of a lattice model for wood.

This is illustrated in Figure 3.2 where the bundles of wood micro-structure are represented by tubular beam elements connected by a Lattice of springs. This approach allows, material defects such as knots, cracks, and grain angle deviations to be incorporated directly into the model. Particularly for a material such as wood, where the heterogeneities cover such a wide range of length scales. A lattice model offers several advantages over conventional continuum representations. The model can firstly represent the material in a physical way. Material elements are arranged in a Lattice that mimics the structure of the real material. Local variations in fiber orientation, shakes and checks, and other heterogeneities or discontinuities can be represented directly in the Lattice. Secondly, the changes in micro-structural features that arise from damaged inducing mechanisms can be handled explicitly through broken elements. Thirdly, and most significantly, the modelling approach establishes a computational link between the bulk behaviour of the material to the micro- and meso-level features causing such a behaviour. Damage, defects, or other micro-structural features are linked to changes in material stiffness and strength. The model can capture damage patterns that result form arbitrary loading states along with the bulk load-deformation response. The benefit of this modelling approach is that it can simultaneously mimic both material damage phenomena and bulk material properties. In this way, the errors produced by homogenising a heterogeneous medium in order to discretise it to permit a finite element solution are resolved. The obvious drawback to the Lattice simulation approach, as with the morphology-based methods, is the extremely large computational requirement for simulation of even a simple wood member [Ric05].

In order to reduce the computation time of the Lattice generation the amount of neighbours to be considered can be adjusted by the user as seen in Figure 3.3.

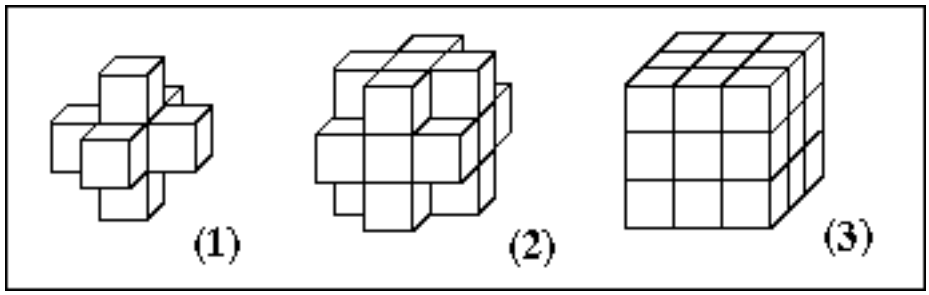


Figure 3.3: Voxel adjacency's in 3D discrete space: (1-faces) the six voxels that are 6-adjacent to the voxel at the center, (2-edges) the eighteen voxels that are 18-adjacent to the voxel at the center, (3-vertices's) the twenty six voxels that are 26-adjacent to the voxel at the center [KCY93].

3.2.2 Fracture Mechanics

In order to fracture the objects ideas based on nonlinear and linear elastic fracture mechanics were applied [2]. The linear elastic fracture mechanics method starts from an initial crack and propagates through the material depending on the strength of the material and direction of the force. This idea has been incorporated into the shatter functions of the plug-in involving forces. The algorithm first searches for the nearest point (InternalCell) in relation to the force endpoint and radius and then propagates from that position depending on the specified force.

3.2.3 Other Algorithms

A further challenge of this masters project was to approximate the actual surface once the internal structure was created. One possibility was to use the well known Marching Cubes algorithm [LC87]. However, one of the limitations of the plug-in is that each face has to be created with four vertices. When using the Marching Cubes algorithm in some of the possible combinations the surface intersects with the cube, creating surface patches which need more than four vertices. Hence, the Marching Cubes algorithm could not be used. The approach taken to approximate the original surface are based on the Marching Cubes algorithm. The algorithm checks which points of a cube are closest to the surface. These points are then moved along a vector depending on how many connected faces belong to the point.

The odd-parity rule [FvDFH96] was used, in order to determine if a point is inside or outside of the object. The rule states that exterior points to an object have an even number of crossings with the surface. Whereas interior points have an odd number of crossings, assuming that the given object is closed [FvDFH96].

For the force shatter radius effect outlined in section 4.4.5, a method had to be used to identify if a given point is inside the force region. Given the fact that a force is of spherical shape, the implicit function test was used to identify the relation between a point and a force [BW97]. The formula used for $point(x, y, z)$ and force with $center(x_0, y_0, z_0)$ and $Radius(R)$ was $f(x, y, z) = 1 - (\frac{x-x_0}{R})^2 - (\frac{y-y_0}{R})^2 - (\frac{z-z_0}{R})^2$

3.2.4 Class Design

With the initial idea of implementing fracture of wood into the plug-in, the complexity of the actual classes involved became obvious very early on. The final class design can be viewed in Appendix A. As the internal structure was build around a voxel based lattice the **Lattice** and **InternalCell** class were introduced.

[2] For an in depth explanation of both methods see [SLG03, And95, Ric05]

Where the InternalCell represents a voxel, the name InternalCell was introduced as the width, depth and height of the actual cell are changeable and therefore give the impression of an actual cell. As neighbouring InternalCells are built from the same vertices the **CellCorner** and **CellWall** class were introduced. This allowed any corner to be moved and to automatically update the neighbouring cells. To enable this functionality each CellCorner would need to know to which CellWall it belongs and *vice versa* each CellWall needed to know which CellCorners it includes. This was provided through the use of the $friendly$ functionality of the C++ language.

During the implementation phase the notion of a **ShatterManager** was created. As the plug-in allows different methods of shattering these options needed to be controlled from a single point in the actual design. Therefore, the ShatterManager is responsible for creating, initialising and reinitialising any of the shatter options. To make the shattering process more transparent the super class **ShatterPiece** was introduced. By specifying the relevant methods as virtual, the ShatterManager class did not have to know which type of ShatterPiece it was dealing with. The sub classes developed to extend the ShatterPiece are **ShatterPieceRandom** and **ShatterPieceForce**.

```
class ShatterPiece {
    public :
        ...
        virtual bool evaluate() = 0;
        virtual bool select(InternalCell *cell,
                            MFloatVector &direction) = 0;
        ...
}
```

The classes that involve the actual Maya plug-in are firstly the **PlugMain** to register the nodes, secondly the **ShatterGenerator** the main node doing the shattering, thirdly the **ShatterOutputLocator** which gives feedback to the user once the shattering is completed and finally the **ForceLocator** representing the forces in the viewport.

3.2.5 Development Environment

The plug-in was developed for Maya 6.0.1 and Maya 6.5. It had to be developed for the two different versions of Maya because of the problems encountered during the implementation. (See Section 4.7.2 and Appendix C) In addition, it was developed for Mac OS X [X05] and Linux[Lin05], by using two different Makefiles to compile and link the classes. On Mac OS X, XCode[X05] was used as the programing environment and Kate [Edi05] on the Linux platform. The plug-in was compiled and linked using g++ version 3.3[gg05].

Subversion [sub05] was used for version control of the source code. The subversion server was set up at the author's home which allowed seamless moving between both the Linux (University) and Mac OS X (Home) environment.

The source code was documented using doxygen [dox05] and is provided as HTML documentation on the CD handed in with this thesis.

Given the complexity of the plug-in and almost infinite combinations of user actions, use cases for the most specific and most common actions were specified. These can be seen in the UseCaseTestingResults in the Documentation folder on the CD. Through these use cases, a certain amount of quality and usability is ensured. During the various tests of the plug-in and the acceptance test documented in the UseCaseTestingResults document, bugs and unexpected problems arouse. These have been documented in the KnownIssuesBugReport also viewable on the CD. It should be noted that only the major bugs have been documented.

Chapter 4

Implementation

4.1 Introduction

This chapter gives a detailed insight into the workings of the shatter plug-in. Before explaining each part in detail, some of the initial ideas are discussed and the key motivation behind them is explained. The main motivation for this specific project was to give every visual effects artist the possibility of using a different shattering tool other than the internal Maya Shatter effect. One of the major differences between this approach and the Maya Shatter is the possibility of actually influencing the shatter piece outcome through the use of forces and the ability to specify the internal cell structure, which can be used to mimic brick structures, cell structures or normal voxel structures.

A further desired feature was to specify the detail of the cell structure. The initial idea was to have three options. The first would be a crude voxel structure, the second would be a voxel structure in which the different sizes of the cell walls could be changed and the third option of detail would be a Dodecahedron structure as this kind would represent the natural structure of wood at its best. However, after seeking advice from Prof. John Vince it was decided to leave it open as an extension to the plug-in, given that the creation of a Dodecahedron structure would increase the development time unnecessarily. The second approach was implemented as it includes the first option an provides the artist with a good influence in the design of the internal cell structure.

4.2 Internal structure

Before turning to the actual creation of the Lattice structure it had to be decided whether to shatter the original mesh and then transform the generated pieces based on the transformations applied to the original mesh, or to shatter the resulting mesh after it has been transformed into its final position. Maya provides the possibility of passing the so called outMesh attribute or the worldMesh attribute of a MeshShape node. However, tests using either attribute did not provide a satisfactory result and no one consulted knew which would be the best option. For example, when passing in the outMesh attribute and evaluating the mesh using MSpace::kWorld, everything was working as expected except rotations which where ignored. The result of various tests on the order and methods to use in order to evaluate any given mesh data in its final world position are shown in Algorithm 1.

Algorithm 1 World mesh methods

Connect the .worldMesh plug of the mesh node into the ShatterGenerator .inMesh plug.
Get the MDataHandle of the MArrayDataHandle:

```
MArrayDataHandle inputMesh = data.inputArrayValue(inMesh,&stat);
...
MDataHandle meshElement = inputMesh.inputValue(&stat);
```

Get the MObject of the MDataHandle and initialise the MFnMesh object:

```
MObject meshObject = meshElement.asMeshTransformed();
MFnMesh mesh(meshObject);
```

Now all method calls to the mesh object are performed in the space specified:

```
mesh.allInternsections(...,MSpace::kWorld, ... );
```

The following section will explain the algorithm used in the second approach to create the internal Lattice structure. The first approach and its main problems are explained in section 4.7.1. During the mid term presentations Dr. Ian Stephenson suggested to try a different approach. With this approach the bounding box of the object was filled with the Lattice structure without testing if the InternalCells are inside or outside of the object. While creating the Lattice structure the neighbour connectivity of the InternalCells was created as well.

4.2.1 Lattice creation

The Lattice structure was initially created by using a 3D array that was extended to hold the amount of elements according to the tessellation steps specified by the user.

```
InternalCell *array[tessStepX][tessStepY][tessStepZ];
```

The optimal distances in which to place the actual points in the Lattice were calculated based on the tessellation steps and the distance between the minimum and maximum point of the objects bounding box.

```
double tessX = fabs((minP.x-maxP.x) / tess[0]);
double tessY = fabs((minP.y-maxP.y) / tess[1]);
double tessZ = fabs((minP.z-maxP.z) / tess[2]);
```

The code constructing all Lattice points can be seen in Appendix D.1. As the points in the Lattice were calculated from the min point of the given bounding box and the tessellation values, the InternalCells were generated based on Figure 4.1.

The number at each corner specifies the order in which the CellCorners were created and added to the InternalCell CellCorner vector. The expression following the number specifies the index to the relevant point in the Lattice point array.

Following the InternalCell creation the algorithm then had to determine the actual CellType of the InternalCell, specifying if it is either on the surface, inside, or outside of the given object. In order to determine the CellType, the odd-parity rule [FvDFH96] as stated in section 3.2.3 was used. Each point of the InternalCell was tested against the given mesh and the parity check was stored. Thus, if all the points had an even parity

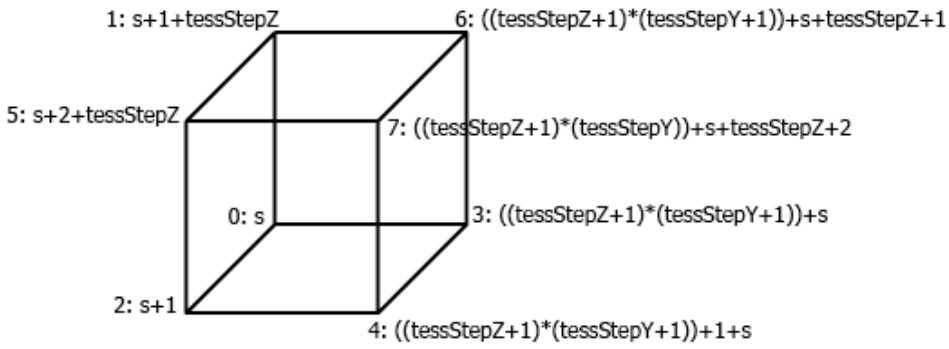


Figure 4.1: InternalCell creation.

check, the InternalCell was outside, if the parity check was all odd, the InternalCell was inside and if there were mixed cases the InternalCell was on the surface.

In order to polygonise the surface at a later stage the neighbours needed to be set up. Each InternalCell keeps a reference to the ID of each of its neighbours using 26 connectivity, as shown in Figure 3.3. On InternalCell creation these references are initialised to -1. Therefore the loop checking the CellType was also used to initialise the neighbour connectivity. As only the InternalCells of CellType inside where considered for neighbour connectivity, the mesh object was filled with InternalCells [1]. The result can be seen in Figure 4.2.

This meant that depending on the position of the InternalCell all the neighbours needed to be checked. Given that all InternalCells were stored in an 3D array these neighbours could be accessed quickly through just adding 1 or -1 to the relevant X, Y, and Z position. However, given that the Lattice was in the shape of the actual bounding box of the object special, tests needed to be made in order to determine if the current InternalCell checked was either:

- One of the eight corners
- On one of the eight edges
- On one of the six faces
- Or internal

These test were necessary as otherwise the border conditions of the array could be overstepped. These adaptions to the initial approach (section 4.7.1) increased the speed to half the time. A summary of the algorithm used to create the Lattice structure is shown in Algorithm 2.

[1] Note: adding a view adaption to the code base would create a plug-in that voxolates any given object.

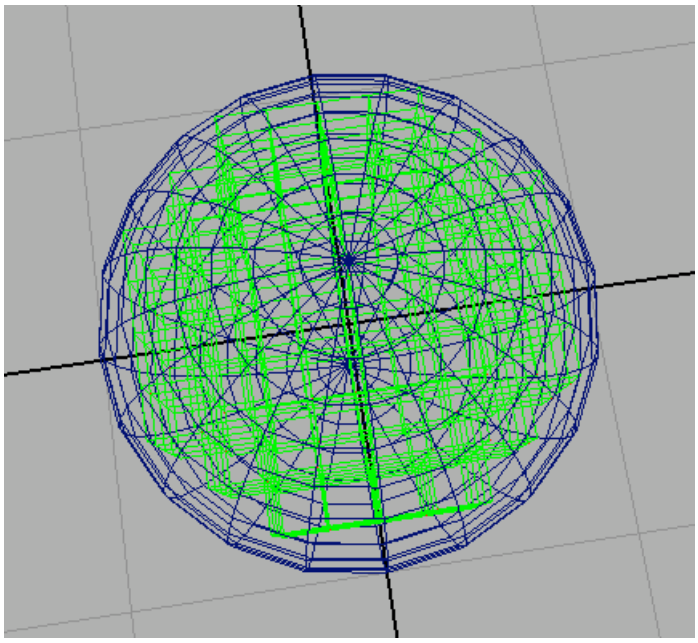


Figure 4.2: Object filled with InternalCells of CellType inside.

Algorithm 2 Lattice creation

Get the bounding box $minP, maxP$ and tessellation $tessStepX, tessStepY, tessStepZ$ steps from the user for the given object.
Create the Lattice, check whether it already exists for the given object based on the meshId.

- If yes check whether any of the tessellation values have changed:
  - If yes call $reset(), createCellStructureNew(mesh, minP), approxSurface(mesh)$
  - If no, stop to save computation time.
- If no:
  - Create a new Lattice and set the initial values. Then call $createCellStructureNew(mesh, minP), approxSurface(mesh)$

4.3 Surface approximation

During the phase of the implementation three different approaches were identified and test versions implemented [2].

1. Marching Cubes
2. From inside
3. Cubes on surface

[2] It should be noted that most of the examples in the papers featured in the Bibliography use proprietary software like qhull [Qhu05] to approximate the resulting surface.

The Marching Cubes algorithm seemed initially the most promising, but as stated in section 3.2.3 the problems encountered, especially the limitation of having four vertices per face, did not lead to a Marching Cubes based solution. The alternative approach of trying to approximate the surface based on the InternalCells that are of CellType surface, resulted in too many test cases to be created, as the states of the neighbours had to be included in order to move the CellCorners into the right direction. The best results were given if only considering the InternalCells of CellType inside with no direct neighbours. Algorithm 3 shows the actual procedure of approximating the surface.

Algorithm 3 Approximating the surface

For all InternalCells:
Check each CellWall if they are of CellType surface:

- If yes for each CellCorner of that CellWall get the normal of all connected CellWalls $NormalN = N + (\sum CellCorner.CellWall.Normal)$. Than execute the $getHit$ method of the Lattice class passing the $mesh, CellCorner, N$.

The $getHit$ method fires a ray from the position of the CellCorner in the direction of the normal passed and stores the closest intersection of that ray with the given mesh. The CellCorner is moved to that position.

Even though this method produced the least artifacts, the results remained unsatisfactory. These artifacts were a result of using normals with a value of 1 according to the relative face. For example: for CellWalls in the X direction $N(1,0,0), N(-1,0,0)$ for CellWalls in the Y direction $N(0,1,0), N(0,-1,0)$ and for CellWalls in the Z direction $N(0,0,1), N(0,0,-1)$. The problem that arouse was that the InternalCells are not unit cubes and therefore moving the CellCorner according to these normals produced intersecting faces. The solution was to create the normals based on the length of the face in each direction. The length of a face is equal to the tessellation value for that direction. Hence, for CellWalls facing the X direction $N(tessX,0,0), N(-tessX,0,0)$, facing the Y direction $N(0,tessY,0), N(0,-tessY,0)$ and finally for facing the Z direction $N(0,0,tessZ), N(0,0,-tessZ)$. Figure 4.3 shows the improvements: the chicken on the left is approximated with the old approach, whereas the chicken on the right is approximated with the new approach and less tessellation, hence the legs are missing.

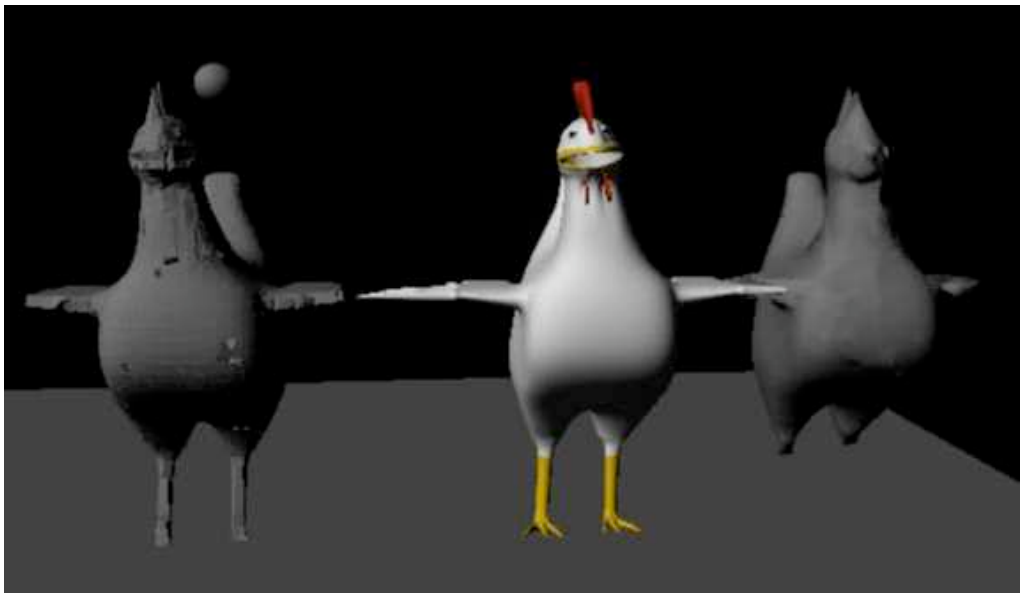


Figure 4.3: Old versus New approximation method

4.4 Shattering

During the phase of the project four different shatter options were developed. The four types were chosen because they represent some of the functionality requested by the contacts in industry. They said that they would appreciate to have an alternative to the Maya shatter tool and to break an object based on some force or a specific impact. The four shatter effects implemented are:

1. Random shattering
2. Force shattering
3. Force random shattering
4. Force radius shattering

4.4.1 Shattering design

Before explaining each of the four shattering effects in detail, this section will give a brief overview of the shattering design. In order to make the shattering process more transparent and to reduce the complexity of the code, the ShatterManager and ShatterPiece class were designed as explained in section 3.2.4. This design greatly reduced the amount of calculations done from the actual ShatterGenerator, as it only had to deal with getting the user information and passing it on to the relevant classes. The main information the ShatterManager is relying on is the aforementioned Lattice. The ShatterManager uses this Lattice and shatters only the pieces that are of CellType surface or inside. The ShatterManager itself provides a method for both initialising any of the four shatter types and executing them. The main sequence of events is shown in the sequence diagram of Figure 4.4.

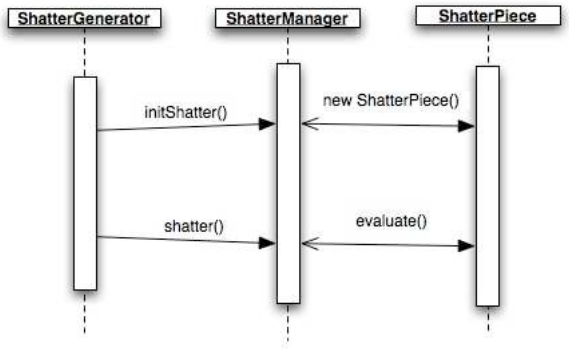


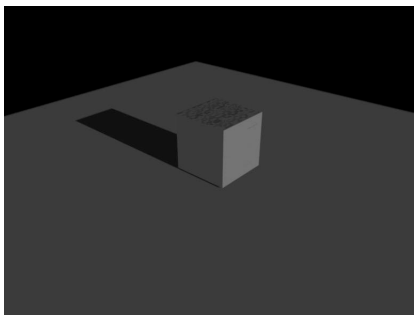
Figure 4.4: Sequence diagram of shatter events.

In general, both random and force shattering are built around the principle of selecting initial InternalCells and then checking if the neighbours already belong to a different ShatterPiece. If this is the case then it is possible to break that connection to the neighbour using the $breakConnection$ method of the ShatterPiece class and mark that CellWall as cracked surface. Otherwise, add this neighbour to the ShatterPiece and continue. To prevent infinite loops each ShatterPiece contains a std::list storing all the neighbours that have not been checked for their neighbours (currentList) and a std::vector of the InternalCells already processed(shatterCells). Whenever an InternalCell was selected from the currentList, it was first removed from that list and its neighbours were checked and if appropriate added to the currentList. After all neighbours were checked the InternalCell was added to the shatterCells std::vector. The ShatterPiece was evaluating its neighbours only until the currentList was empty, as this indicated that no direct neighbouring InternalCells were available.

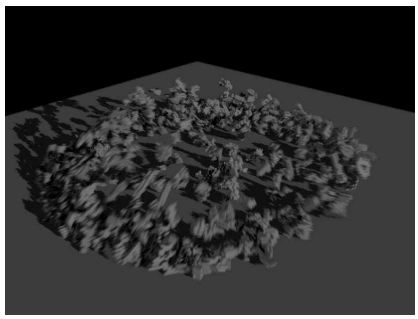
4.4.2 Random shattering

The random shatter effect is based on the idea of the standard Maya shatter effect. The only values the user specifies are the tessellation of the InternalCells and how many pieces are required. The $initRandomShatter()$ function of the ShatterManager loops according to the number of shatter pieces and randomly selects any of the InternalCells inside the object. When selecting an InternalCell a ShatterPieceRandom gets initialised and the InternalCell passed into the constructor.

Following this method call the $randomShatter()$ function gets executed. This method loops through all the generated ShatterPieces calling their $evaluate()$ method each time in the loop. The $evaluate()$ method returns true if the ShatterPiece evaluated still has unchecked neighbour cells. As shown in Appendix D.2, the check for testing whether to continue evaluating the ShatterPieces is done through applying the boolean or (||) operator to all of the results. Only if all of the $evaluate()$ method calls return false the loop will be terminated. As stated in the above section the evaluate method checks all of its neighbours depending on six connectivity as shown in Figure 3.3. Initial test showed that when selecting the neighbours in the same order, the ShatterPieces had the same appearance. Hence, the order with which to select the neighbouring cells was randomised. Figure 4.5 shows a cube shattered with the random shatter algorithm.



The initial cube



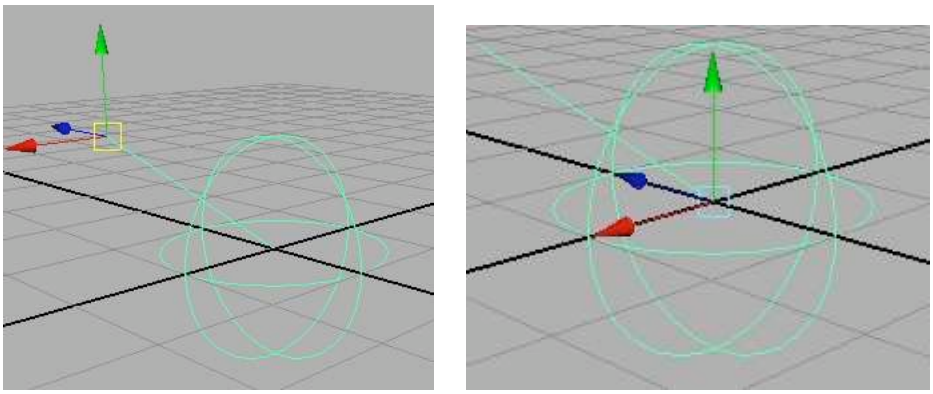
The cube randomly shattered and animated with dynamics

Figure 4.5: Randomly shattered cube.

4.4.3 Force shattering

The force shatter effect is based on the statement by the contacts in industry that to be able to split / break and object by specifying a force would be beneficial. In order to specify a force the ForceLocator was created. The ForceLocator is a visual representation of a force with an impact radius, a start and end position of the force and the impact radius shown in figure 4.6.

To generate the direction vector of the given force, the ShatterGenerator simply uses $dir = forceEnd - forceStart$ to identify the vector. The simple force shatter option just takes the force and splits the geometry into two pieces. To achieve this affect the two InternalCells closest to the end point of the force are calculated using Algorithm 4 on the next page.



The ForceLocator node The ForceLocator with the Manipulator

Figure 4.6: ForceLocator

Algorithm 4 Identifying the closest cells to force

Create four place holder variables: $distance, distanceTwo, cellid, cellidtwo$
For each InternalCell:

1. Calculate the center of that InternalCell.
2. Calculate a vector between the center and the force end position $vec = center - forceEnd$
3. Calculate the distance of the vector $testDistance = vec.length()$
4. $if(testDistance < distance)$ store $distance = testDistance$ and $cellid = InternalCell- > cellId$ otherwise test $if(testDistance < distanceTwo\&\&testDistance > distance)$ store $distanceTwo = testDistance$ and $cellidtwo = InternalCell- > cellId$

The resulting two InternalCell IDs are then passed to the ShatterManagers $initForceShatter()$ function. To generate the cracks the following idea was used: generate two ShatterPieceForce given as initial InternalCells the two InternalCells closest to the force end position. Continue to select the neighbouring InternalCells that are on the direction vector until an InternalCell of CellType surface is found (the crack passed to the other side). A unit cube and the two cracks generated are shown in Figure 4.7.

To identify which neighbouring InternalCell to select next, an almost identical approach to finding the closest InternalCell to the force end position was used. The current InternalCell computes its center point and moves that point along the direction of the force. The InternalCell then checks each of its neighbours. The number of neighbours to check is specified by the user. The InternalCell closest to its center point and the newly created center point gets selected as the next InternalCell along the direction of the force. Initial tests resulted in straight cracks through the given object. However, as this plug-in was suppose to generate organic looking cracks the force direction was slightly altered each time. To alter the force the vector from the InternalCells center to the neighbouring InternalCell selected to be next was computed. The computed vector was then added to the direction vector of the force. To ensure the crack was moving along the direction of the initial force the alternation vector was multiplied by -1 for every second function call. As stated above when an InternalCell of CellType surface was selected the ShatterPieceForce signals to the ShatterManager that it has finished and it will no longer be asked to check for the next neighbour.

After the crack paths are generated, the ShatterPieces alternately select the neighbours as in the random shatter algorithm. In the same order according to the neighbour connectivity specified by the user, as shown in Appendix D.3.

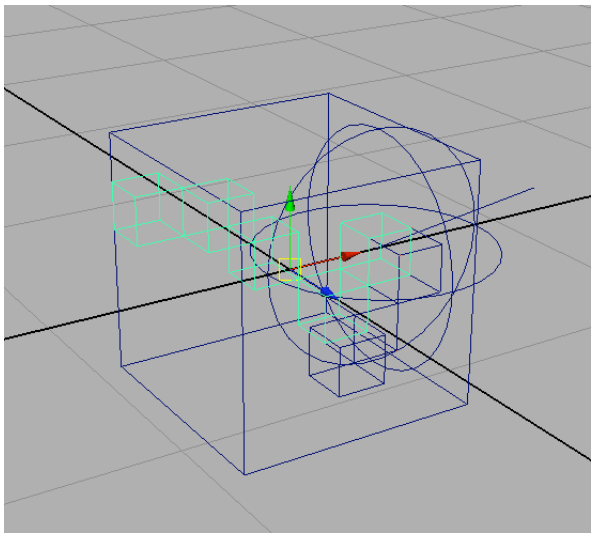


Figure 4.7: Crack generated through force shatter

4.4.4 Force random shattering

The shatter effect "ForceRandomShatter" is a combination of the two shatter effects mentioned above. To generate the ShatterPieces, the given force is taken and a number of InternalCells as specified by the user is selected following the random shatter approach. After all the InternalCells are selected, the $initForceRandomShatter()$ method of the ShatterManager creates the crack paths using the same approach as the force shatter. Through this process the object shattered looks as if the whole piece was shattered by the force direction.

4.4.5 Force radius shatter

The last shatter effect, "ForceRadiusShatter", provides the user with the possibility of creating effects such as glass shattering or any other form of impact shattering. This is achieved is through not only testing for the two nearest InternalCells to the force end position, but to test for all the InternalCells that are inside the force radius. To identify if an InternalCell is inside the force radius, the implicit sphere function specified in section 3.2.3 was used. To make the plug-in more usable and not limited to one force, only the $affectedPieces$ attribute was added to the ShatterManager. The attribute was added as an std::map with the InternalCell ID as the key and the corresponding force as the element, $std :: map < int, MFloatVector > affectedPieces$. Thus, if a ShatterPiece was already influenced by a force the stored force direction was added to the additional force direction. The main problem with this approach to shattering, is that it is hard to predict how many InternalCells are included in the radius region of the force. If the user specifies a number for the shatter pieces greater than the actual number of InternalCells in the radius region an error will occur. However, this error will be prevented and an error message will be displayed to the user asking him to either reduce the number of shatter pieces or to increase the tessellation.

The process of shattering is the same as that for the random force shatter except that the initial pieces are not selected randomly, and the affectedPieces provide the key to get the relevant InternalCell and pass the connecting force. Figure 4.8 shows an example of a glass wall where the center is shattered with a force and one hundred pieces.

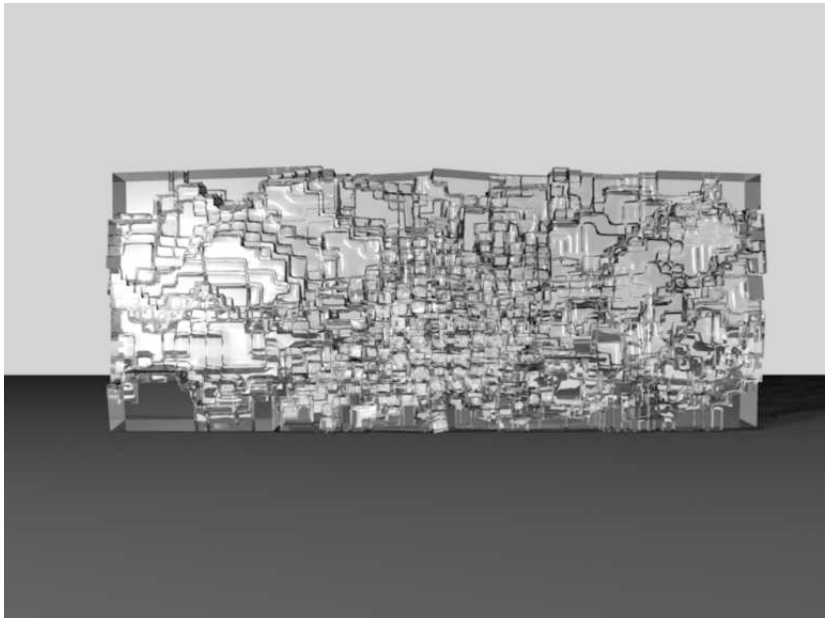


Figure 4.8: Glass wall shattered with force shatter radius.

4.5 Polygonisation

In order for Maya to generate a new mesh the following attributes are needed [May05, Dav03]:

- **numVertices** number of vertices
- **numPolygons** number of polygons
- **vertexArray** point (vertex) array. This should include all the vertices in the mesh, and no extras. For example, a cube could have the vertices: { (-1,-1,-1), (1,-1,-1), (1,-1,1), (-1,-1,1), (-1,1,-1), (-1,1,1), (1,1,1), (1,1,-1) }
- **polygonCounts** array of vertex counts for each polygon. If for example, the cube had 6 faces, each of which had 4 vertices, the polygonCounts would be {4,4,4,4,4,4}.
- **polygonConnects** array of vertex connections for each polygon. For example, in the cube, we have 4 vertices for every face, so we list the vertices for face0, face1, etc consecutively in the array. These are specified by indexes in the vertexArray: for example for the cube: { 0, 1, 2, 3, 4, 5, 6, 7, 3, 2, 6, 5, 0, 3, 5, 4, 0, 4, 7, 1, 1, 7, 6, 2 }
- **parentOrOwner** parent of the polygon that will be created
- **ReturnStatus** Status code

Hence, to create all the ShatterPieces the following Algorithm 5 was used:

Algorithm 5 Polygonisation

For each ShatterManager:
For each ShatterPiece ((*begin)) of the ShatterManager:
Create three placeholders and execute $polygonize()$:

```
...
MPointArray vertices;
MIntArray polyCount;
MIntArray polyConnects;
MIntArray surfaceFaceIds;
(*begin)->polygonize(c,vertices,polyCount,polyConnects,surfaceFaceIds);
...
```

The $polygonize()$ method of the ShatterPiece calls the $polygonize()$ method of each of the InternalCells in turn.
The $polygonize()$ method of the InternalCell checks if any of the direct neighbours have a neighbour id of -1. If yes the method adds the corresponding values to the given $vertices, polyCount, polyConnects$:

```
if(backNbour == -1) {
  addFace(vertexArray,polyCount,polyConnects,0,1,6,3);
  if(backNbourType == SURFACE) {
      surFaceIds.append(faceId);
      localFaceIds.append(faceId);
      faceId++;
  } else {
      faceId++;
  }
}
```

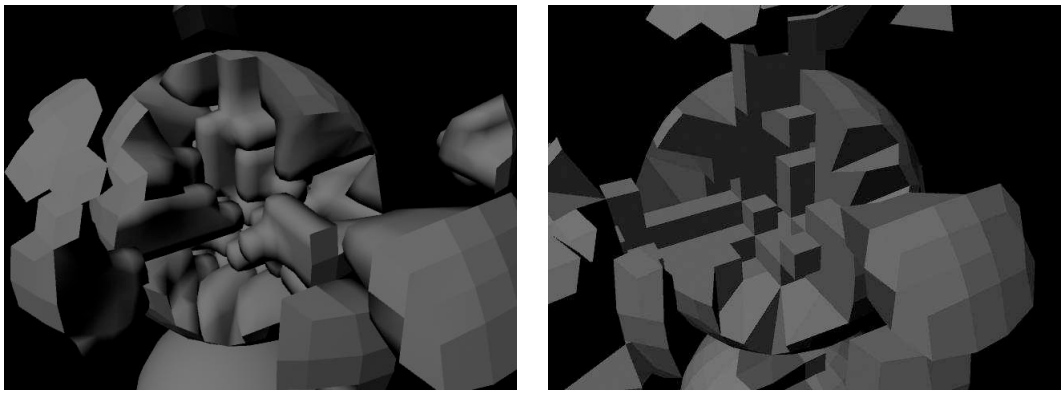
As explained in the introduction section 4.4.1 the $breakConnection()$ is responsible for splitting up the InternalCells. Hence, the complexity of creating new surfaces is overcome by simply setting the relevant neighbour id to -1.

Initial tests showed that the resulting ShatterPieces where cubical in shape. Therefore, the option was developed to smooth the internal edges of the ShatterPieces. The smoothing was achieved by specifying which edges of the resulting mesh where “hard” or “soft”. When applying a Polysmooth node to a mesh node the Polysmooth node checks for soft edges and “smoothes” them as shown in Figure 4.9.

To achieve this option the following algorithm had to be implemented. Whenever Maya generates a new mesh the edges are automatically “smoothed”. The desired affect was to only smooth the internal parts of the ShatterPiece. Hence, all the edges that are connected to an outside face needed to be remembered. Whenever a polygon gets added to a mesh, Maya automatically increases the faceIds of that mesh. Thus, when adding the vertices to mesh object the current face id needed to be stored as shown in the last part of algorithm 5. After adding all the faces of the ShatterPiece an extra loop had to be introduced to set the external edges as “hard”. The algorithm simply generated an MItMeshPolygon and compared the index of each polygon with the index stored in the surfaceFaceIds array. If they are the same, a polygonal face that is placed on the outside is found. Finally the edge ids of the polygonal face needed to be retrieved using the MItMeshPolygon method $.getEdges()$ to set the relevant edge smoothing to be false.

4.6 Computation of the plug-in

The aforementioned sections explained each part of the Shatter plug-in in detail. This section provides an insight into the actual process that is behind the shattering process, including some aspects of the usability



Internal cracks smoothed Internal cracks not smoothed

Figure 4.9: Polysmooth operation

of the plug-in. The whole process of a single computation of the ShatterGenerator node is explained in the following Algorithm 6.

Figure 3.1 shows the Dependency Graph after the shatter force command has been executed. The figure shows the connections that are made when any of the force shatter commands are executed. For the force shatter to work the ShatterGenerator needs to know how many forces are acting on the object to shatter, as well as the corresponding directions.

To prevent the ShatterGenerator to recompute the Lattice and ShatterManager each time any of the attributes changed, some precautions had to be made. In general, for the Lattice to recompute the only attributes that needed to be changed were in relation to the three tessellation values. In order for the ShatterGenerator generator to recompute the attributes in relation to: "number of shatter pieces, change in forces, change in shatter mode, change in number of neighbours to consider, and if the Lattice has updated" had to be changed. Thus, to prevent both the Lattice and ShatterManager to recompute each time, the ShatterGenerator keeps a reference to all the Lattice and ShatterManagers in two std::vectors.

```
std::vector<Lattice*> latticesVector;
std::vector<ShatterManager*> shManager;
```

Hence, for each new computation the ShatterGenerator checks if a Lattice and ShatterManager have already been created for any of the input meshes. To check if there are existing Lattices and ShatterManagers the plug-in makes use of the fact that once a mesh is connected to the input MArrayDataHandle of the meshes the index at which the connection is made will not change. Hence, the ID of both the Lattice and ShatterManager gets set to the ID of the mesh index in the MArrayDataHandle. Thus, on re-computation of the ShatterGenerator the ShatterGenerator checks if there is a Lattice and ShatterManager for the given mesh index and reassigns them to the corresponding elements of either std::vector. If no Lattice and ShatterManager exist, the ShatterGenerator creates the new elements and assigns the mesh index to both IDs and adds them to the latticesVector and shManager. In a future release both the std::vector<ELEMENT*> will be changed to std::map<int,ELEMENT*> where the mesh index is simply a key to the corresponding element. This means less time has to be spend when checking for existing IDs in the vectors.

The problem with this approach is that extra checking needs to be done at the end of processing the Lattices and ShatterManagers. If a mesh gets removed the Lattice and ShatterManager for that object will still exist and produce output if asked. Therefore, for each run the ShatterGenerator keeps track of the current meshes producing output and performs a check removing those inactive.

Algorithm 6 ShatterGenerator compute

Setup the MComputation Interrupter, to enable interruptions with the ESC key.
Check if the re-computation request affects any of the values that change the computation of the Shatter-Generator node.
Get the user specified values for the tessellation($tess$), shatterType($sType$), neighbour connectivity($nConnect$), edge smoothing ($smooth$), and the number of shatter pieces($sPieces$).
In addition get the forces if $sType$ is not of shatter type random. For the forces the algorithm needs to retrieve all the forces connected to this ShatterGenerator therefore the start, end position as well as the radius of the force have been set up as MArrayDataHandles to support more than one force. Hence, two MFloatPointArrays and one MFloatArray are created and the forces retrieved.
For all the meshes connected:
Generate a MFnMesh object and retrieve the current mesh data.
Check if $sType$ is of shatter type random or any of the forces influence this mesh. If not, check if a ShatterManager exist for this mesh index and reset it (not to produce "ghost shapes").
Generate the BBox.
Generate the tessellation and number of tessellation steps (section 4.2.1).
Check if a Lattice and ShatterGenerator have been created for this mesh already.

- If yes, retrieve the pointer to the already created Lattice and ShatterGenerator:
  - Check if the tessellation values have changed. If yes update the values and call $reset, createCellStructureNew, approxSurface$.
  - Check if any of the values causing the ShatterManager to reevaluate have been changed. If yes reinitialise the values changed and call the relevant initShatter and Shatter function according to $sType$
- If no:
  - Create a new Lattice and set the values and call the relevant methods.
  - Create a new ShatterManager and set the values and call the relevant methods.

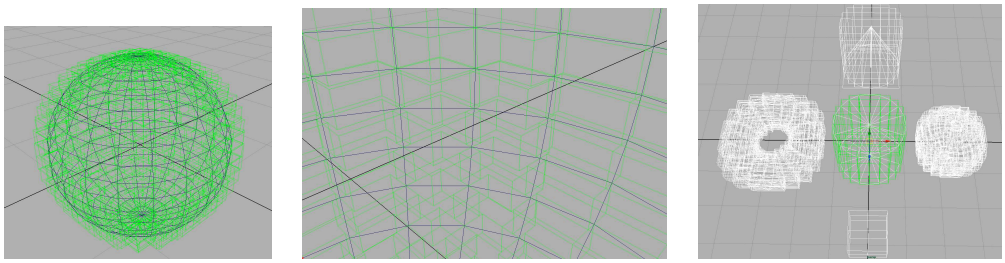
Prepare the outputArrayHandle and get the MArrayDataBuilder for it.
Check that only ShatterManagers that have been update or are relevant for the output meshes add meshes to the output.
For each ShatterManager:
For each ShatterPiece:
Create the place holders as specified in the polygonisation algorithm 5. Perform the edge smoothing test if specified by the user $smooth$.
Set all the output values clean and the computation has completed.

4.7 Problems

This section outlines some of the major problems encountered during the project. Among these were the inexperience with Maya API programming, no documentation about bugs in the API, and limited help available for questions about the Maya API. Furthermore, the initial plan was to include fracture mechanics which build on the physical properties of a given material. This could have been achieved if the aforementioned Maya API problems would not have occurred. Hence, the actual shattering of the object is not as advanced as anticipated, but the underlying structure is built to include these for further development.
A further problem with the plug-in is in relation to human computer interaction (HCI), because in its current state the work-flow of the plug-in is based on the authors experience which could be different from actual production work. It would have been of benefit for HCI if actual tests would have been conducted including artist to test the plug-in and to document their comments.

4.7.1 Initial approach

Most of the time was spent approaching both the internal structure and approximating the surface. The initial idea was to create the internal structure based on the actual mesh. By creating a bounding box for each face, the internal structure should have been built from the resulting intersections of the neighbouring bounding boxes. Even though this worked efficiently for the neighbouring cells there was no easy and efficient way to check when cells from the other side were intersecting. An approach for generating the bounding boxes of each face was implemented and can be seen in figure 4.10.



Sphere | Sphere close up | Primitives

Figure 4.10: Initial cells structure creation with bounding boxes

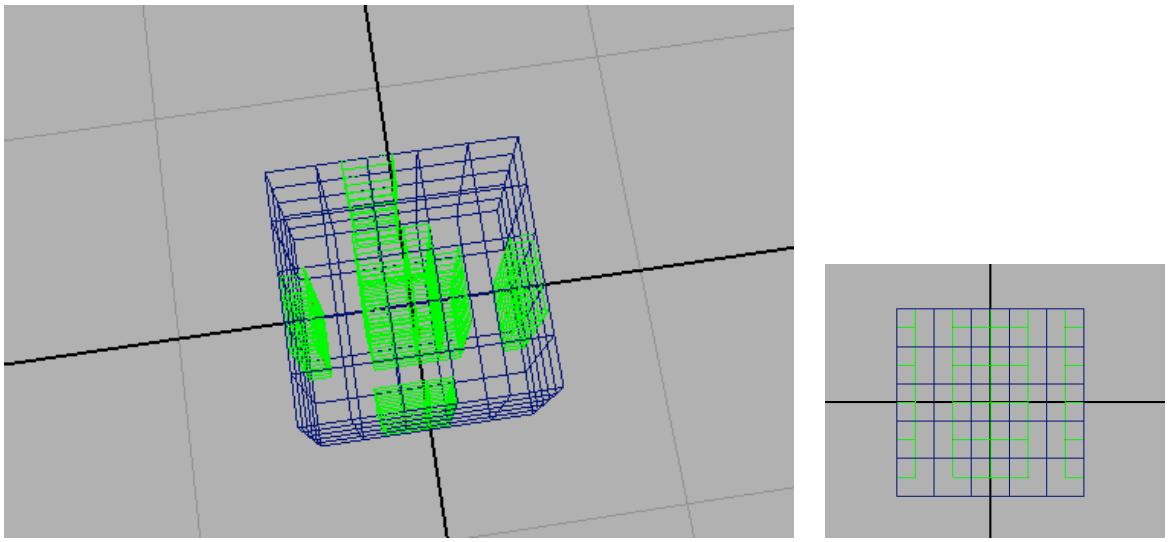
As this approach did not produce the desired outcome a second approach was tried. This time the idea was to use the bounding box of the object and sample the space to a user given tessellation. While tessellating, the algorithm checked if a point lies within the object through the odd-party rule outlined in section 3.2.3. After the tessellation, the algorithm checked each point whether it is part of an InternalCell that exists given to the X, Y and Z tessellation values. If the InternalCell exist check if its already created. In order to create the neighbour connectivity, create the immediate neighbours above, in front, and to the right of this InternalCell. If they exist, set the neighbour type according to the connection and move on to the next points. In order to check if an InternalCell exists, the algorithm checks if the MPointArray includes any of the corner points of the InternalCell. If it does, the InternalCell is created. After checking if the neighbours exist, the method evaluates the CellType and creates the CellWalls of the InternalCell.

This approach became very intensive in terms of computation, as all the test cases had to be made. As one of the criteria of the contacts in industry was performance, the approach in section 4.2.1 was identified and implemented. The method for the old approach $Lattice.createCellStructure()$ has been left in the Lattice class for reference. The improvements through the final Lattice creation makes the implementation twice as fast.

4.7.2 Maya API

This section will mention some more general problems as well as various technical problems encountered when using the Maya API. Then main difficulty was testing and debugging the Maya API. Given that the plug-in was developed on Mac OS X and Linux, no advance debugging method of setting breakpoints and stepping through program calls was found. Hence, all the debugging was done through plain printing to the Maya Script Editor, the console or by analysing stack traces once Maya crashed. Most of the print statements used are left commented in the source code to illustrate the various steps needed to take to find bugs in the plug-in.

Another more general aspect is the fact that the Maya API is itself not free of bugs. Especially when working on the mesh approximation, the method provided to check intersections did not provide accurate results. As stated in Appendix C , the MFnMesh::intersect method provides faulty results when used with a polygon cube as the bounding box has exactly the same dimensions as the object itself. The error which occurred can be seen in Figure 4.11.



Unit cube perspective Unit cube side view

Figure 4.11: MFnMesh::intersect problem

As Alias gave the advice that the problem was related to the size of the bounding box being the same, the bounding box was made slightly smaller. This improved the method but did not fully resolve the problem as shown in Figure 4.12.

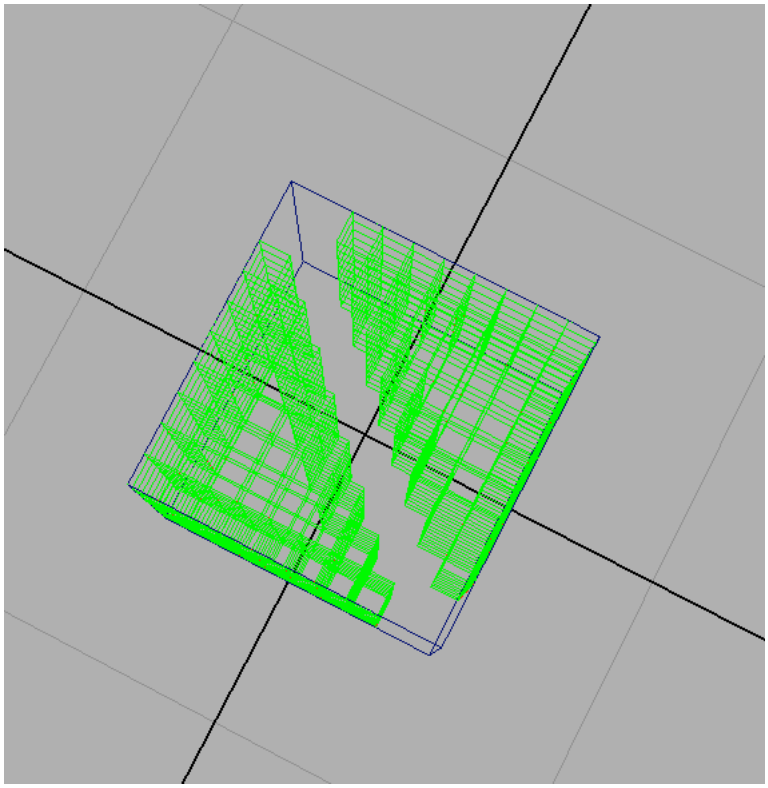


Figure 4.12: MFn::intersect problem with smaller bounding box.

The advice given by the Alias Maya support team was to use the Maya 6.5 API as it introduces three new intersections methods: $anyIntersection(), allIntersections(), closestIntersection()$. These methods resolved the problem but raised another problem as that Maya 6.5 was not available at Bournemouth University during the duration of this masters project. Hence, two versions of the plug-in were developed concurrently. While working in Maya 6.5 time was also spend to make the plug-in work on 6.0.1 using the adapted method to do mesh intersection calculations.

Additional problems were created by the use of two different platforms, Mac OS X and Linux, the plug-in was developed for. Interestingly, these were in relation to casting between the short and int datatype. For example, the enum attribute of the Maya API gets set as short. When retrieving the attribute value and using the $asInt()$ method casting problems occurred on the Mac OS X platform. This must have been because of the big Indian - small Indian difference of both platforms. For example, 1 retrieved using $asInt()$ resulted in 65536 (-1).

The final problem occurred in relation to mesh creation. Ideally the plug-in should have created the resulting shatter pieces without the user having to perform an additional step. The problem here is that the Maya API data model is based around a node only knowing its inputs and outputs. This means that a node cannot create other nodes and setup connections between them. One solution would have been to create the mesh nodes and to connect them to the ShatterGenerator output in the mel commands. However, this creates another problem as in some shatter cases it is not clear how many shatter pieces there are after the ShatterGenerator has computed. The decision was made to use the ShatterOutputLocator to signal to the user when the ShatterGenerator has finished computing by drawing the resulting Lattice points as red points in the viewport. Whenever the artist sees these points he/she can execute the mel scripts, $createShatterPieces(int\$smooth)$ or $recreateShatterPieces(int\$smooth)$ to create and connect the resulting shatter pieces.

For a detailed insight into the bugs and problems that were faced during the development please refere to the KnownIssuesBugReport.pdf located in the Documentation folder on the CD handed in with this master project. Moreover, conditions that raised awareness of the bugs can be viewed in the UseCaseTestingResults.pdf in the folder mentioned above.

Chapter 5

Conclusions and future work

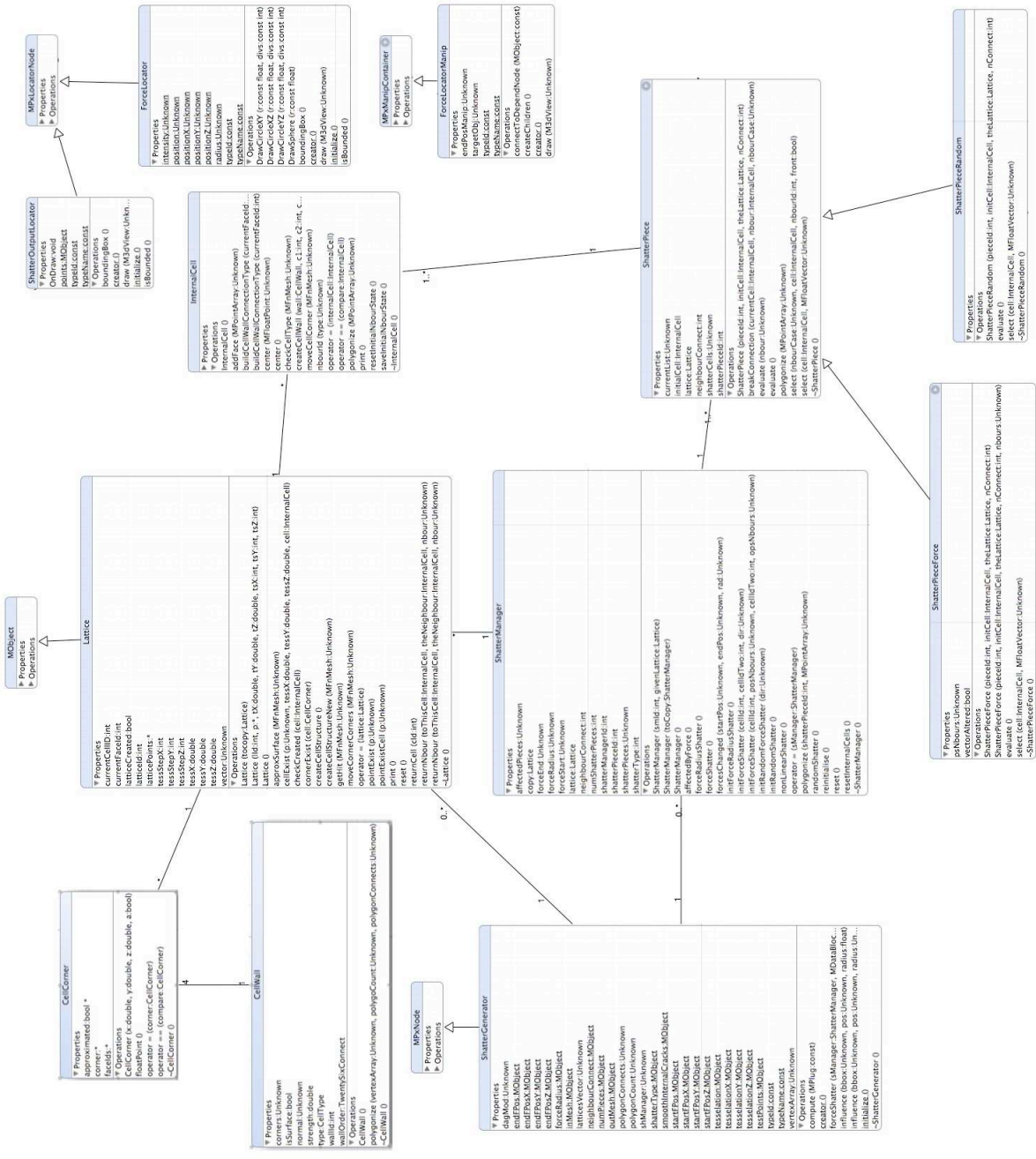
The initial aim this project was to simulate the fracture mechanics of wood and to develop these into a plug-in in Maya. The project did not achieve these objectives. This is mainly due to the problems outlined in section 4.7. The delays caused by these problems left no time to actually implement real fracture mechanics into the plug-in. However, some major achievements were made and an immense knowledge about plug-in development using C++ and the Maya API was gained. The actual code base is of high quality and uses object orientated features. Moreover, the class design proved very robust as it did not change significantly for the final design and no problems was encountered that would doubt it correctness. Furthermore, the contacts stated that they would benefit from a tool that would allow them to shatter any geometry based on a force which is achieved in three different modes in the plug-in. Through a collaboration with Apiwat Srimongkolkul of the MA Digital Effects it could be proved that the plug-in can be used in a production. For this project a given model in the shape of a woman was taken and shattered into one thousand pieces. The plug-in performed this task without any problems. However, the collaboration raised one very important aspect when shattering in production. Given the machine power available, the simulation of one thousand pieces becomes a very slow and tedious task as simulating these pieces over a duration of 300 frames takes around 30min to cache. In the end, only five hundred pieces were used in the simulation as only the reduction in numbers of pieces made the simulation possible.

The main motivation for creating this plug-in was to provide the artist with a shattering option other than the standard Maya Shatter function. In comparison to the Maya shatter effect this plug-in proves far more advanced when a great number of pieces need to be generated. For example, generating one thousand pieces off a unit cube takes roughly 30 min with the standard Maya shatter. The Shatter plug-in, on the other hand, produces one thousand pieces in under one minute complete with mesh creation. Both tests were executed on a dual G5 2.4Ghz.

This masters project lays down the fundamentals for generating an internal cell structure of any given polygonal object which can now be taken further and implementations based on actual fracture mechanics can be included. Additional work is needed to approximate the surface, as artifacts are still generated if the geometry is complex. One of the initial ideas was to implement the shattering by drawing lines on the surface of the object and then to generate the volume and cracks based on these lines. This approach got positive feedback from contacts in industry and was supposed to be implemented as an add-on at a later stage. Time limitations meant that it could not be included in the implementation.

Appendix A

Class Diagram



Appendix B

Email - Contacts

Subject:

Re: Tips zu einem Masters Project

From:

Carsten Kolve - R&D MPC

Date:

30. April 2005 17:35:17 GMT+01:00

To:

Hannes Ricklefs

Hi,sorry fuer die spaete antwort - bin im Moment etwas im Stress .. ok, zu deinen fragen. Hier ein paar Punkte auf die wir hier achten

a) scalierbarkeit: es sollte moeglich sein den algorithmus interaktiv anzuwenden um nachher, wenn man mit der basissimulation zufrieden ist, das ganze mit mehr details aber fast gleicher animation berechnen zu lassen

b) flexibilitaet: du willst dein plugin speziell auf das brechen von holz hin entwickeln - das ist schoen und gut nur leider schraenkt es den anwendeungsraum doch extrem ein. alle moeglichen arten von effekten dieses types haben aehnlich eigenschaften also solltest du diese gemeinsamkeiten allgemein halten. so kann man dann mit verschiedenen parametern unterschiedliche effekte erziehlen. zwar kannst du deine arbeit immer noch holz fokussieren - aber warum solltest du dir nicht die option fuer andere stoffarten offenhalten? es ist nicht wesentlich mehr programmieraufwand aber dein plugin wird wesentlich flexibler.

c) kontrolle: idealerweise will man als td alle moeglichen parameter einer simulation kontrollieren koennen. und wenn man das mal nicht will sollte das plugin mit einer simulation oder emulation die noetigen entscheidunden selbst treffen. Beispielsweise koennte das fuer eine Cracking-Shattering system bedeuten das das system die bruchstellen* aufgrund physikalischer eigenschaften des objektes selbst bestimmt, * ein einfacher algorithmus musterbasierte bruchstellen generiert * der nutzer selbst, zb mit einer texturemap, bruchstellen bestimmt * kombination der obigen punktevergiss nicht die zeitkomponente!

d) schnittstellen: es kommt haeufiger vor als man denkt das programme entweder zweckentfremdet werden ode nutzer an internen werten interessiert sind von denen man nie vorher dachte das sie von allgemeinem interesse sind - also gib alle sinnvollen internen werten mit einem interface an die aussenwelt: zb waeren collision points /normals interessant um sie in ein particle system zu stecken

e) pipeline: mach die gedanken wie die objekte / animation weiterverwendet werden - wie wird was gerendert? caches? shading? uv maps? etc.

f) benutzerfreundlichkeit: ein kleiner mel menu set fuer die gaengigsten arbeitschritte sollte schon da sein, dokumemtier * die software und den workflow den du dir ausgedacht hast * die wichtigsten mel scripts die mit objekten arbeiten - diese solltenvollkommen losgeloest von einem ui nutzbar sein so das ein td sie in seinen eigenen scripten wieder verwenden kann* bilder, tutorials, icons sind des td's freund :)

Alles sehr allgemein gehalten, aber es sollte schon mal ein paar einblicke in prinzipien auf die wir wert legen geben. wenn du noch mehr fragen hast, dann melde dich!carsten

Subject:

Re: Quick Question

From:

Andy Hayes - Animal Logic

Date:

4. Mai 2005 07:16:50 GMT+01:00

To:

Hannes Ricklefs

Hi,

...

In regards to a wood fracturing plugin, in principle it would be great to have a program crack things based on stress associated with the geometry and from possible impact with it. There is a distinct lack of shattering tools. Some realism would be good, but always remember that a good tool has to be totally controllable, which is something of a oxymoron when it comes to simulating physics. Having the ability to specify fracture points, direction of fracture, additional forces to 'guide' shattered elements and timing, etc would be impressive.

I liked the group project by the way - good shattering!

Hope things go well.

Andy

Subject:

Re: Tips zu einem Masters Project

From:

Jan Walter - R&D The Mill London

Date:

11. Juli 2005 13:34:11 GMT+01:00

To:

Hannes Ricklefs

Hallo Hannes,

ich bin's nochmal ...

Splittern von Holz klingt interessant. Da gibt's bestimmt ein paar interessante SIGGRAPH papers dazu. Wenn Du Informatik studieren wuerdest, dann wuerde ich sagen, die Integration in Maya (oder andere Pakete) ist nicht so wichtig und der produzierte Code der Software ist wahrscheinlich sowieso nicht produktionstauglich. Das entspricht zumindest meiner Erfahrung. Wir haben damals unsere eigene Firma gehabt und einer von uns hat Diplomarbeiten an der Technischen Uni Berlin betreut. Es ging mehr darum,

herauszufinden, ob bestimmte Techniken wert sind, weiterverfolgt zu werden, oder nicht. Wenn der Student ein Standalone oder ein Plugin gebastelt hat, das funktionert, war das mehr als ausreichend. Wenn man die Technik in der Produktion verwenden will, muss man sowieso alles neu schreiben ...

Tja, wenn das fuer Informatiker gilt, dann sollte es fuer Studenten in Bournemouth auch gelten, allerdings sehe ich ein, dass ihr etwas naeher an der Produktion orientiert seid als z.B. die TU. Also solltest Du rausfinden, was wichtiger ist (fuer Bournemouth): Die Integration oder ein neuartiger (?) Algorithmus. Falls es die Integration ist, solltest Du eventuell mit Rob van den Bragt reden, was er fuer wichtig halten wuerde, um das Plugin zu benuzten. Er ist Hollaender, spricht also auch Deutsch, wenn Du ihn auf Deutsch anschreiben willst. Englisch tut's natuerlich auch ... Du kannst auch Robert Kolbeins fragen. Beide sind sehr erfahrene Maya Benuzter. Vom Programmieren her wuerde ich mir eher eine saubere Integration in Maya wuenschen und weniger einen neuartigen Algorithmus. Als Informatiker waere es genau umgekehrt, ich wuerde von einem Informatiker erwarten, dass er einen Vergleich verschiedener Algorithmen vornehmen wuerde und zumindest ansatzweise mehrere Algorithmen implementieren wuerde. Die Qualitaet der Software wuerde ich weniger hoch bewerten wie die Qualitaet des theoretischen Vergleichs (Vor- und Nachteile der verschiedenen Algorithmen). Naja, das ist zunaechst erst mal alles, was ich zu sagen habe ...

Ich hoffe meine Email kommt nicht zu spaet und hilft Dir etwas ...

Gruss,Jan

Appendix C

Email - Alias support

Subject:

Alias case ID 1-2047432

From:

Alex Ang

Date:

28. Juli 2005 13:34:11 GMT+01:00

To:

Hannes Ricklefs

Hi Jonathan,

I will look into the case about tesselating a mesh with internal points andthen using MFnMesh::intersect. Usually, I find that my test code will workon cubes but not more complex geometry. However, it looks like a cube may bean interesting boundary case since its bounding box will correspond exactlywith the geometry.If possible, can you send me the code you are using for tesselating the meshand doing the intersection test? Also, you mentioned that you can sendpictures explaining the problem. Please send those, too. Finally, if you areusing Maya 6.5, will using one of the new intersection methods(anyIntersection(), allIntersections(), closestIntersection() that isapplicable give you different results?

Alex Ang

Support Product Specialist

Alias Platinum Membership Delivery

210 King St. East, Toronto ON, Canada M5A 1J7

Contact information:

http://www.alias.com/support/contact

Subject:

Re: Alias case ID 1-2047432

From:

Alex Ang

Date:

28. Juli 2005 13:34:11 GMT+01:00

To:

Hannes Ricklefs

Hi Hannes,

Thanks for the code and images. I'm trying out the code, but I need a couple of bits. I would like to know what the value of "NORMALY" is set to and what the type of "vertexArray" is (it appears to be an MPointArray).

Also, it appears you are using this code within a custom node. If that's the case, then there's a few lines that, although may work, is considered bad design for a custom node. For example, you cannot use the node to get selected objects and process an object from the selection list. A custom node can only take data from its incoming connections, compute and set the result to its output attributes. In your case, the mesh to be tesselated must ideally come from some attribute, say inMesh, and it will be in the form of mesh data, not an actual mesh node or DAG path. For example:

MDataHandle inMeshHandle = data.inputValue(inMesh);

MObject meshData = inMeshHandle.asMesh();

MFnMesh mesh(meshData);

I believe you can use the meshData object on MItMeshVertex to figure out the bounding box. However, you cannot find the mesh's transform or transformation matrix, since what you have is mesh data and this is separate from the mesh node it came from (and if you try to follow connections to the mesh node you will be breaking the design principles for custom nodes).

With mesh data, you won't be able to perform worldspace operations (MSpace::kWorld) on MFnMesh or MItMeshVertex, although in your case you are only doing object space ops. So you should be fine.

Alex

Subject:

Re: Alias case ID 1-2047432

From:

Alex Ang

Date:

28. Juli 2005 13:34:11 GMT+01:00

To:

Hannes Ricklefs

Hi Hannes,

World space operations are impossible if the mesh data comes from attributes like outMesh or anything that stores the mesh in object space. If you use worldMesh, then you have the world coordinates right off the bat.Glad to know that your code is working. I recommend using the new 6.5 intersect methods as they are faster.I can allocate unique identifiers for your nodes, so I have given you a block of 64 IDs in the following range: 0x0010CEC0 - 0x0010CEFFI can allocate more for you if needed.

Alex

From: Alex Ang [mailto:alexang@alias.com]

Sent: Monday, August 08, 2005 10:33 AM

To: 'Hannes Ricklefs (d1132332)'

Subject: Alias case ID 1-2157310

Hi Hannes,

You can actually store pointers or references in your node, as long as you fully own the objects underneath them (like your own classes or data structures). I believe what you mentioned only applies to objects owned by Maya, which you have handles to via MObjects. Maya won't have any knowledge of objects you fully own, so you can keep pointers or references to them as long as you know how to manage them.

On the other hand, objects owned by Maya and accessed as MObjects can be moved by Maya without your knowledge, so at some point, an MObject you have can become invalid. This is why we tell Maya API programmers not to hang on to MObjects and assume they will always point to a valid Maya object. For example, in one of your node's methods, you freshly retrieve an MObject. That MObject will be valid for the duration of that method call. However, the next time that method is called, the MObject can become invalid, so you should always re-retrieve the MObject.

If you really want to store your custom objects as attributes, then you will derive a class from MPxData. Please consult the online docs as well as the examples in the devkit for more details.

Alex

Appendix D

Code - Examples

D.1 Lattice Points

```
double x = minP.x;
for(int donex = 0; donex < tessStepX+1; donex++) {
    double y = minP.y;
    for(int doney = 0; doney < tessStepY+1; doney++) {
        double z = minP.z;
        for(int donez = 0; donez < tessStepZ+1; donez++) {
            MPoint point(x,y,z);
            latticePoints->append(point);
            z+=tessZ;
        }
        y+=tessY;
    }
    x+=tessX;
}
```

D.2 Evaluating ShatterPieces

```
bool running = true;
std::vector<ShatterPiece*>::iterator begin;
std::vector<ShatterPiece*>::iterator end;
while(running) {
    begin = shatterPieces.begin();
    end = shatterPieces.end();
    bool stop = false;
    while(begin != end) {
        bool runningTest = (*begin)->evaluate();
        stop = (stop || runningTest); begin++;
    }
    if(!stop) {
        running = false;
    }
}
```

D.3 Evaluate ShatterPiece according to neighbour connectivity

```
bool ShatterPiece::evaluate(TwentySixConnect nbourCase) {
    if(currentList.empty()) {
        return false;
    } else {
        InternalCell *cell = currentList.front();
        int nbourId = cell->nbourId(nbourCase);
        if(nbourId != -1) {
            InternalCell *nbour = lattice->returnCell(nbourId);
            if(nbour->shatterPieceId == -1) {
                nbour->shatterPieceId = shatterPieceId;
                currentList.push_back(nbour);
            } else if(nbour->shatterPieceId != shatterPieceId) {
                breakConnection(cell,nbour,nbourCase);
            }
        }
        return true;
    }
}
```

Bibliography

[AHS96] R J Astley, J J Harrington, and K A Stol. Mechanical modelling of wood microstructure, an engineering approach. In *IPENZ Conference*, 1996.

[BW97] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[CMHT02] Mark Carlson, Peter J. Mucha, R. Brooks Van Horn, and Greg Turk. Melting and flowing. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 167–174, New York, NY, USA, 2002. ACM Press.

[Dav03] Gould David. *Complete Maya programming*. Morgan Kaufmann, 2003.

[dox05] doxygen. *www.doxygen.org*, 2005.

[Edi05] Kate (KDE Advanced Text Editor). *kate.kde.org*, 2005.

[FOA03] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. *ACM Trans. Graph.*, 22(3):708–715, 2003.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM Press.

[FvDFH96] Foley, van Dam, Feiner, and Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley, second edition, 1996.

[gg05] gnu g++. *www.gnu.org*, 2005.

[HR90] H.J. Herrmann and S. Roux. *Statistical models for the fracture of disordered media*. Elsevier Science Publishers B.V(North-Holland), 1990.

[KCY93] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics, sidebar: Fundamentals of voxelization. *IEEE Computer*, 26(7):51–64, 1993.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[Lin05] Redhat Linux. *www.redhat.com*, 2005.

[May05] Alias Maya. *www.alias.com*. Alias, Head Office, Toronto, Canada, 2005.

[NFJ02] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728, New York, NY, USA, 2002. ACM Press.

[OBH02] James F. O'Brien, Adam W. Bargteil, and Jessica K. Hodgins. Graphical modeling and animation of ductile fracture. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 291–294, New York, NY, USA, 2002. ACM Press.

[OH99] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 137–146, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[PKA+05] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutr, Markus Gross, and Leonidas J. Guibas. Meshless animation of fracturing solids. *ACM Trans. Graph.*, 24(3):957–964, 2005.

[Ric05] Hannes Ricklefs. Personal research - identifying an approach to implement wood fracture for computer animation/vfx. Technical report, Poole, England BH12 5BB, Unpublished, 2005.

[SLG03] Ian Smith, Eric Landis, and Meng Gong. *Fracture and fatigue in wood*. Wiley, 2003.

[sub05] subversion. *subversion.tigris.org*, 2005.

[SWB01] Jeffrey Smith, Andrew Witkin, and David Baraff. Fast and controllable simulation of the shattering of brittle objects. *Computer Graphics Forum*, 20(2):81–90, June 2001.

[TF88] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1988. ACM Press.

[WKDL+05] Wittel, Falk K., Dill-Langer, Gerhard, Kroplin, and Bernd-H. Modeling of damage evolution in soft-wood perpendicular to grain by means of a discrete element approach. In *Computational Materials Science*, volume 32, pages 594–603, 2005.

[X05] Mac OS X. *www.apple.com*. Apple Computer, Head Office, California, USA, 2005.