Thomas Lefley
Girton College
tl364

Computer Science Part II Dissertation

Dynamic Volumetric Fragmentation: *Mesh to voxel transformation for optimised physics-based interactions*

Monday 11th May, 2015

Thomas Lefley, Girton College

Dynamic Volumetric Fragmentation: Mesh to voxel transformation for optimised physics-based interactions

Computer Science Part II, 2015

$9,950$ *Words*

Project Originator
Thomas Lefley
Project Supervisor
Flora Ponjou-Tasse

Original Aims

The project aimed to investigate the use of volumetric representations of rigid objects in physics-based simulations of model destruction.

Work Completed

A voxelisation algorithm has been implemented to transform rigid meshes to their volumetric representation in addition to a voxel-to-mesh method. A method for object destruction based on an interpretation of the physical forces involved has also been achieved.

Special Difficulties

None.

I Thomas Lefley of Girton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Thomas Lefley
Date: Monday 11th May, 2015

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem 1

1.3 Solution 1

1.4 Related Work 4

1.4.1 Fragmentation 4

1.4.2 Mesh/Voxel Conversion 4

1.4.3 Related Uses 4

2 Preparation **5**

2.1 Refinement 5

2.2 Resources Used 5

2.3 Required Understanding 6

2.3.1 Voxelisation 6

2.3.2 Fragmentation 6

2.3.3 Meshing 6

2.4 Requirements Analysis 7

2.5 Planning 7

2.5.1 Pipeline 7

2.5.2 Class Structure 8

2.6 Conclusion 8

3 Implementation **9**

3.1 Process 9

3.2 Preprocessing 9

3.2.1 Assumptions 10

3.3 Voxelisation 11

3.3.1 Data structures 11
3.3.2 Triangle Phase 12
3.3.3 Propagation Phase 12
3.3.4 Progress 12
3.4 Physical Destruction 14
3.4.1 Data structures 14
3.4.2 Finding Generating Points 14
3.4.3 Partitioning 15
3.4.4 Finding Islands 15
3.4.5 Progress 18
3.4.6 Advantages 18
3.5 Intermediate Processing 19
3.5.1 Separating Border Voxels 19
3.6 Mesh Generation 20
3.6.1 Data Structures 20
3.6.2 Mesh Partitioning 20
3.6.3 Marching Tetrahedra 21
3.6.4 Progress 24
3.7 Post-processing 26
3.7.1 Physics 26
3.7.2 Game-object Generation 26
3.8 Result 26
3.9 Conclusion 28

4 Evaluation **29**
4.1 Comparison 29
4.1.1 Advantages 29
4.1.2 Disadvantages 30
4.2 Empirical Evaluation 34
4.2.1 Metrics 34
4.2.2 Unity3D Overhead 35
4.2.3 Simple Object 36
4.2.4 Complex Object 39
4.2.5 Multiple Objects 43

4.2.6 Results 47
4.3 Requirements Met 47
4.4 Feasible Continuations 48
4.4.1 Accuracy 48
4.4.2 Parallelism 48
4.5 Conclusion 49

5 Conclusions **51**

Bibliography **53**

A Expanded Tables **55**

Chapter 1

Introduction

1.1 Motivation

Physics simulation is one of the key components of a modern game engine. The more accurate the simulation is, the more immersive games built with the engine are likely to be. If the world around the player does not react as expected then it is difficult to become immersed in the game environment.

1.2 Problem

The most common representation of a solid object in a physical simulation is a rigid polygonal mesh. This format stores the vertices and faces which define the object's surface, with volume information being implicit. This is optimal for operations which require only the surface information, such as rendering, but presents issues if we wish to reason on the object's interior, as we do when fragmenting under physical forces.

It is possible to fragment an object without knowledge of its interior by simply partitioning the space in which the object exists and refining any partitions which have faces in them. However, as no volumetric information about the object is being used, the fracturing is approximate. A further limitation is that most implementations using similar approaches are restricted to convex objects or must first decompose any concave objects into constituent convex meshes [13].

Overall, while fragmentation from only surface information is efficient and produces reasonable results, the physical accuracy is limited and this project instead proposes a method which uses the object's volumetric information.

1.3 Solution

Taking an object as a three dimensional scalar field of mass, we simplify the problem of fragmenting it on the applied physical forces. For each point inside the object we determine which fragment that point should belong to based on the object's physical properties.

When only the boundary is stored, determining whether any given point is inside of an object

is non trivial. Consider a series of faces defining a 2D object. Then to label any given point as inside the object we need to check that it is 'inside' of all of the faces [2]. Furthermore, this only holds for convex shapes, if there exist any concave sections then we could have a point which is 'outside' of a face but inside the 2D object. See Figure 1.1.

Calculating for all points, whether they are inside the object before physical interaction can reduce the simulation time complexity discussed above. We also gain a constant time lookup for checking if any given point lies within the object. This process is known as binary solid voxelisation, where a binary voxel is a point in three dimensional space with an assignment based on whether it is inside or outside of a given object [15].

This project proposes a pipeline for simulating physical destruction, focussing on fragmentation, consisting of the following stages.

1. Objects are voxelised
2. Each voxel is assigned to a fragment algorithmically in such a way as to emulate the physical properties of the object
3. A rigid mesh is built for each fragment from its constituent voxels

For example, if a bullet is fired at a brick wall, both the bullet and wall will be rigid meshes when the bullet is shot. On collision, both the wall and bullet will be voxelised. Assuming the bullet consists of only one voxel, no destruction calculations will be performed on it. Using the force of impact and physical parameters of the wall, it will be calculated which voxels should belong to which fragments. These fragments will then be transformed back into rigid meshes and the correct movement vectors will be applied. This is illustrated in Figure 1.2.

This process will be referred to as *Dynamic Volumetric Fragmentation (DVF)*.

Figure 1.1: The green area shows all points 'inside' line (a). The red areas shows all points 'outside' line (a). Point (b) is 'outside' line (a) but inside the defined concave shape.

Figure 1.2: A simple rigid mesh projectile is fired towards another rigid mesh (a). On collision the destructible body is voxelised (b) depending on the force of the projectile and physical parameters of the objects. It is determined which voxels will detach, shown in red. Meshes are reformed around all separate bodies and physical forces are applied (c).

1.4 Related Work

1.4.1 Fragmentation

One existing method for object destruction is a static approach. Fragments are generated for every destructible object when the objects are created and then substituted for the object on collision. This does not allow the result of the collision to be in any way dependant on the forces involved in the collision and so does not produce realistic results.

Matthias Müller, Nuttapong Chentanez and Tae-Yong Kim propose a method for the dynamic destruction of objects in real time. The method uses 'volume approximate convex decompositions' to fragment objects which are composed of convex pieces using Voronoi decomposition of space [13].

1.4.2 Mesh/Voxel Conversion

Binary voxelisation of polygonal objects is a memory efficient method for defining occupied space and can be achieved in real-time on modern GPUs. Both surface and solid voxelisation methods are presented by Schwarz and Seidel [15]. A solid voxelisation algorithm was used as opposed to surface voxelisation as the entire volume of an object is required for the proposed volumetric destruction.

Both marching tetrahedra and surface nets exist as algorithms for constructing 3D surfaces from volumetric data [3] [10]. An existing implementation of marching tetrahedra has been used as part of the remeshing stage of the pipeline as more literature existed concerning the use of this method with Unity3D [16].

1.4.3 Related Uses

Nießner et al present a method for real-time collision detection of patch based objects using mesh to voxel transformations [14]. While their work is on collision detection, it does demonstrate that real-time mesh to voxel conversion is viable for use in resolving physics interactions.

Miguel Cepero is developing a procedural voxel engine Voxel Farm [5]. His work includes volumetric destruction of objects in a manner similar to this project's aims [4].

Chapter 2

Preparation

This chapter will detail the planning and preparation done before the project was undertaken. The refinement of the original idea, as found in the project proposal, as well as the necessary groundwork undertaken will be discussed.

2.1 Refinement

This project has the aim to investigate a new approach to resolving physically induced damage in objects. However, the term damage encompasses a wide range of processes. Furthermore there are many stimuli that could result in physical damage occurring.

It was decided that the scope of this project would be limited to fragmentation as a form of damage. We define fragmentation to be the partitioning of a rigid solid into smaller rigid solids, assumed to be made of the same material as the original object. This process should conserve mass as much as possible, with no loss between the original object and sum of the resultant fragments.

The physical stimulus that this project is constrained to is collision. We focus on simple collisions between two discrete objects where only one object is capable of fragmenting. We use relevant information from the collision such as object masses and impact velocities and use it to construct as accurate an outcome as possible, including ensuring each fragment has an accurate movement vector post collision.

While the simulation conditions have been constrained for the scope of this project, the principle of computing an object's volume in order to accurately simulate the destruction of that object is transferable to wider conditions.

2.2 Resources Used

Unity3D was used as a basic engine with which to work. The engine provided general features such as rendering, rigid mesh physics and game object management which were expanded upon to achieve the more specific required goals. As Unity3D is a widely used commercial engine, any time or space overheads introduced by the engine can be more safely ignored in analysis as they would also be present in other solutions. There also exist other object destruction libraries

written for the engine which have been used for comparative evaluation [12].

A HLSL DirectCompute shader implementing the Schwartz method for solid voxelisation, found in GPU Pro 3 has been adapted for use and a C# driver written [15] [9]. The original library had a C++ driver which handled both the setup and dispatch of the shader as well as rendering. The methods regarding setup and dispatch were analysed and their functionality reproduced in C# using Unity library calls for invoking dispatches.

An existing implementation of the marching tetrahedra algorithm on the CPU for voxel to mesh transformations [16] was used. The original library generated a 3D scalar field from perlin noise and then used this to implicitly define a surface which was then meshed using marching tetrahedra. The perlin noise generating code was removed and the 3D field used by the marching tetrahedra methods exposed as an input.

2.3 Required Understanding

The majority of code for this project was written in C#, a language which required some learning.

2.3.1 Voxelisation

In order to write a driver for the GPU implementation of solid binary voxelisation found, both the method and shader language used had to be understood. The description of the algorithm featured in GPU Pro 3 was read and a tutorial for HLSL, particularly its use with Unity3D, followed [9].

2.3.2 Fragmentation

An understanding of basic physics was required for the resolution of damage as described. This was to ensure that the outcome was as accurate as possible under the involved forces and defined object properties. As the project is more focussed on the Computer Science aspects of this method of damage resolution, the physics calculations used are not so complicated as to truly simulate the real world.

Voronoi methods for fracturing were also studied as these are what are most commonly used for fragmentation in simulations. This is because they produce results that are dynamic rather than predefined[1] and are efficient [13].

2.3.3 Meshing

The marching tetrahedra algorithm was studied for use forming the implicit surfaces defined by the voxels in each fragment. Both a GPU and CPU implementation were found and explored. The CPU implementation was found to be sufficient and used as a starting point for the meshing phase of the pipeline [16].

[1] See Section 1.4.1.

2.4 Requirements Analysis

In order for the project to be considered successful in investigating a new approach for object damage it was decided that the following should be met:

- The solution *must* enable a scene in which an object with up to 65,534 vertices[2] is fragmented in a manner which resembles a real world interaction as closely as possible.
- The solution *must* be object independent. That is any object may be made destructible without retroactively altering the object's definition.
- The solution *must* take into account the defined physical characteristics of the object to be fragmented as well as those defined by the collision when resolving.
- The solution *should* produce fragments with the correct physical properties based on those of the original object and the size of the each given fragment.
- The solution *should* work with both convex and concave objects.
- The solution *could* run in real-time.
- The solution *won't* deal with texture mapping onto fragments.
- The solution *won't* deal with structural integrity.

2.5 Planning

2.5.1 Pipeline

As detailed previously, the destruction process was split into three sections:

1. Voxelisation
2. Partitioning
3. Remeshing

It was decided what each section should input and output ahead of time to ensure two things.

- Different implementations of each section could be easily interchanged.
- Each section would output something that allowed for easy testing of that section.

Figure 2.1 illustrates this decision.

[2]The limit imposed by Unity3D

Figure 2.1: The proposed pipeline.

2.5.2 Class Structure

The decision was made to have driver classes which processed the inputs and outputs to their respective implementation of a pipeline stage. An overarching driver would then manage the passing of output to input between drivers, doing any intermediate data processing required. A delegating driver managing all of the pipeline stage drivers rather than an actual pipeline of driver was chosen as it allows for easier manipulation of data between stages as well as debugging. Switching out different implementations of the stages also becomes easier.

2.6 Conclusion

In order for the implementation of the proposed solution to run smoothly and the requirements to be achieved, it was necessary to become familiar with the technologies, languages and algorithms to be used. The most important part of preparation was devising the pipeline and investigating the existing solutions for the required work in each stage. This was so that implementation could proceed in a sequential manner through the pipeline with knowledge of the upcoming stages allowing for correct design.

Chapter 3

Implementation

This chapter will discuss the stages of development. The starting point, implementation and result of each pipeline stage will be detailed as well as any difficulties faced.

3.1 Process

The process for development was iterative, implementing each pipeline stage in order. This was because each stage required the output of the previous stage to function and, while mock results could have been generated and forwarded, often the easiest way to input interesting data was to generate it from the previous stage because of the size of the data structures being used.

Stages were kept as encapsulated as possible so that any changes that had to be made would not cascade through later stages or require changes to those previous.

Testing and debugging were done on each stage as it was completed to test the performance of that particular implementation and gauge whether changes or a new algorithm were needed. This also improved the ease of finding bugs in comparison with debugging at the end of the pipeline.

3.2 Preprocessing

The simplest way to satisfy the requirement of applying the correct resultant movement vector to each fragment is to perform the fragmentation before the collision takes place. This allows the Unity3D physics engine to properly apply force to each fragment when the incoming particle makes contact.

It is therefore necessary to predict and extrapolate possible collisions before they occur. This

was achieved by adding a bounding sphere to destructible objects and using Algorithm 1.

```
Data: Object to be destroyed: d, Bounding sphere: s, Incoming object: p
Result: Force magnitude exerted on each object: f, Collision point on d: c
Ray r = cast ray from p along p.velocity;
if r intersects d then
    Point c = intersection point;
    Vector n = c.normal;
    n = n*2;
    Double md = d.mass;
    Double mp = p.mass;
    Vector vd = d.velocity;
    Vector vp = p.velocity;
    Double t = Physics.timestep;
    f = Dot(n, (mpvd - mdvp));
    f = f / (md+mp);
end
```

Algorithm 1: Collision extrapolation

This algorithm relies on the following equations which approximately calculate the magnitude of the force which will be exerted on each object[1]:

$$F = \frac{\beta}{t} \quad t = \text{time objects are in contact for} \tag{3.1}$$

$$\beta = \frac{2\hat{\mathbf{N}}_{\mathbf{1}} \cdot (m_2\mathbf{v_1} - m_1\mathbf{v_2})}{m_1 + m_2} \tag{3.2}$$

As the colliding objects are rigid we take the time that they will be in contact for to be one time-step of the Unity3D physics engine.

3.2.1 Assumptions

As we are extrapolating the collision data at the point when they enter the bounding sphere, some assumptions are made.

- The velocity of the two objects involved will not change between this point and any collision
- A third object will not move to obstruct any collision after this point
- The rotation of either object will not change the result of any collision

By tightening the bounding sphere around our destructible object we can take these assumptions to hold for simplicity.

[1] Dylan Kennedy, personal communication, January 14, 2015

3.3 Voxelisation

Originally it was planned for only a portion of the destructible object to be voxelised, depending on the physical forces involved. However it was found that the computational complexity of determining which mesh triangles fell within this portion was equal to that of voxelising these triangles. Therefore, for simplicity, the whole mesh is voxelised.

The solution for voxelisation used is based on an implementation of the Schwartz method found in GPU Pro 3 [15] [9].

This method is made up of two phases, a triangle phase and a propagation phase. Both of these are separate shaders run on the GPU to allow for parallel processing within the phases.

3.3.1 Data structures

The data structures used in this stage of the pipeline are primarily buffers as these can be passed to and from the GPU shaders as input and output.

The inputs to the first shader are the triangles defining the object's mesh. These are stored in two buffers. A 'vertex' buffer stores a list of floating point numbers, each one being an x, y or z coordinate in a triangle's vertex, relative to the position of the object[2]. These values are always grouped such that for each vertex the x, y and z coordinates are adjacent and in sequence. There is no further ordering imposed. A second 'index' buffer stores an integer list of indices into the vertex buffer. This buffer is ordered such that, when divided into strides of three, each index within that triplet refers to the point of a triangle when used with the vertex buffer. This is shown in Figure 3.1.

The output from both shaders and input to the second is a three dimensional array of bits defining which voxels exist. This is stored as a single integer buffer, with each integer encoding 32 voxels. The three dimensional array is flattened, first along the Z direction, then X, then Y. The Z direction is padded such that all of the voxels in a integer will have the same X and Y coordinates. On being outputted from the second shader this flattened array is wrapped in a helper class which allows easy lookup into the array as if it were three dimensional as well as helper functions such as getting the array's size in any of the three dimensions represented.

[2]Therefore the world position is not factored into the vertices.

Figure 3.1: The triangle (c) is represented by three consecutive indices, one for each vertex, in (b) which each point to the beginning of three consecutive floats which define that vertex in (a).

3.3.2 Triangle Phase

During this phase, each triangle in the object to be voxelised is processed in its own GPU thread. First the triangle is translated and scaled by a pair of matrices defined by the object's bounding box and the requested voxel resolution[3]. This transforms object relative world coordinates into 'voxel coordinates'.

The object relative coordinates are defined by taking a tight bounding box around the object, the centre of the box is $(0, 0, 0)$ and the box extends to 0.5 in each dimension. Voxel coordinates define $(0, 0, 0)$ to be a corner of the bounding box and each axis extends to a defined magnitude depending on the voxel resolution desired. This is shown in Figure 3.2.

Triangles are then projected onto the XZ plane and all of the intersecting voxel columns are found[4]. This is shown in Figure 3.3. For all of these columns the first voxel below the triangle, in the Y direction, is set.

3.3.3 Propagation Phase

In the propagation phase we move up the Y axis, XORing each voxel with the one below in the Y direction. This allows us to fill in the solid volume. As we move up, when the first set voxel is met all those above will then become set until we reach another that had been set by the triangle phase. Each Y column is processed in parallel, with 32 columns being processed in parallel on the same thread as we are XORing integers which are encoding 32 voxels.

3.3.4 Progress

Both shaders were taken from the GPU Pro 3 library mostly unaltered[5]. The C++ driver found in the library was used to gain insight on interfacing with the shaders when writing the C# driver used. The completed driver and wrapper for the voxel grid output were heavily inspired by a JAVASCRIPT CPU implementation of voxelisation in Unity3D [6].

[3]The X, Y and Z dimensions of the voxel grid.
[4]Voxel columns extending along the Y axis.
[5]There were methods in the first shader for other, unneeded, forms of voxelisation which were removed.

Figure 3.2: The box (a) represents a voxel space of resolution $7 \times 9 \times 7$. (b) shows the same box in object relative coordinates.

Figure 3.3: The red triangle has been projected onto the XZ plane and the intersecting voxel columns marked in green.

Achieving binary solid voxelisation on the GPU was the first major milestone completed. Several other libraries and methods were tried and tested before a suitable one was found. It also took much more time than anticipated to complete integration of the library which was used.

A scene was constructed where several objects differing in triangle resolution and shape were used for debugging the implementation. Figure 3.4 demonstrates the output with the voxel structure being displayed for debugging.

Figure 3.4: Voxelisation of the 'Stanford Bunny' model, composed of 69,666 triangles. The voxelisation shown is lower resolution than that which is typically used in the final implementation resulting in 1,723 voxels shown in red.

3.4 Physical Destruction

The process of physically destroying an object involves partitioning its volume into fragments. This is done by assigning each voxel to a fragment, which will be referred to as 'colouring'.

The method used is based on Voronoi diagrams as these are commonly used for achieving visually realistic fracturing effects [13]. This choice was made because of time restrictions and the encapsulated pipeline nature of Dynamic Volumetric Fragmentation would allow a more physically accurate algorithm to easily take its place.

3.4.1 Data structures

This pipeline stage takes the voxel grid from the voxelisation stage as an input and outputs a list of 'fragment' classes and an altered voxel grid.

The fragment class stores:

- An integer 'colour' for that fragment, which acts as an identifier.
- A list of points that make up the volume for that fragment.
- A minimum and maximum X, Y and Z coordinate which define the bounding box for that fragment in voxel space.

The altered voxel grid is a three dimensional array of integer values with each one corresponding to the colouring of the voxel in that position.

Duplicate information is returned here as different algorithms further down in the pipeline, primarily the meshing phase, require different input data structures. Some work on the point cloud provided by the fragment objects while others work optimally on the grid where neighbours can be found in constant time.

3.4.2 Finding Generating Points

To fragment the object using a Voronoi diagram, a series of generating points have to be found.

Firstly, the collision point from the preprocessing stage of the pipeline is transformed into voxel space. A number of vectors are then generated from this point with their magnitudes normally distributed within a given radius. The number of points and radius are calculated from both user defined physical properties of the object such as 'brittleness' and 'strength' as well as the collision force. This ensures that the fragmentation is different depending on the physical characteristics of the collision and therefore more accurate.

The vectors have normally distributed magnitudes so that there are more generating points closer to the collision point. This results in many smaller fragments about the point of impact as you would expect.

3.4.3 Partitioning

Given the generating points P_i, a Voronoi diagram is a partitioning of space such that a cell contains all points in space which are closer to P_i than any P_j. This is formally defined by equation 3.3 and shown in Figure 3.5.

$$\text{Given a set } S = \{p_1, p_2 \dots p_n\} \quad \text{A Cell } C(S, p_i) = \{p \in R^d \mid |p - p_i| < |p - p_j|, i \neq j\} \tag{3.3}$$

A fragment object is made for each generating point and all voxels are looped over. For each voxel, all generating points are checked and the closest found. That voxel is then added to the corresponding fragment object, also updating the minimum and maximum coordinates for that fragment, and the voxel is given the correct integer colouring in the new grid.

3.4.4 Finding Islands

A Voronoi partitioning is always guaranteed to produce convex solids when applied to a convex solid. However, as Dynamic Volumetric Fragmentation works on concave models this cannot be relied on. If a partitioning spans a concavity then it may produce unconnected islands as shown in Figure 3.6. Meshing such a partition into a rigid solid would result in floating areas as the physics engine cannot reason within a mesh as seen in Figure 3.7.

To find and remove any islands a flood fill algorithm is used. Algorithms 2 and 3 demonstrate how this works.

```
Data: Dictionary of fragments indexed by colour: d, Greatest fragment colour: c,
      Coloured grid with islands: g
Result: Dictionary of new fragments with no islands: o, Coloured grid with no islands: g
Dictionary<integer, fragment> o;
for Each fragment f in d do
    for Each voxel v in f do
        if g[v.x,v.y,v.z] == f.colour then
            Fragment r;
            r.colour = ++c;
            o.Store(r.colour, r);
            Flood(v.x, v.y, v.z, f.colour, r.colour, r, o);
        end
    end
end
```

Algorithm 2: Flood initiation

For every voxel in every fragment, if that voxel is still coloured with the same value as that fragment then it has not been reached by a previous flood for that fragment. Therefore it is not connected to the already flooded portion of that fragment and should instead be part of a new fragment. As should all connected voxels found by flooding from the voxel.

Figure 3.5: A Voronoi diagram [7].

Figure 3.6: The blue Voronoi cell creates two unconnected partitions of the concave object.

Figure 3.7: If the unconnected ears and body are a single mesh then the physics engine cannot act on the ears causing them to fall.

```
Data: Vector to flood from: v, Old colouring: f, New colouring: r, Dictionary of fragments
      indexed by colour: o, Coloured grid with islands: g
Result: Dictionary of new fragments with no islands: o, Coloured grid with no islands: g
Queue<Vector> neighbours;
HashSet<Vector> visited;
neighbours.Enqueue(v);
visited.Add(v);
while neighbours is nonempty do
    Vector deq = neighbours.Dequeue();
    int x = deq.x;
    int y = deq.y;
    int z = deq.z;
    g[x, y, z] = r;
    colouring.Add(v);
    for i from -1 to 1 do
        for j from -1 to 1 do
            for k from -1 to 1 do
                int xi = x + i;
                int yj = y + j;
                int zk = z + k;
                if i == 0 AND j == 0 AND k == 0 then
                    continue;
                end
                if xi out of bounds in g OR yj out of bounds in g OR zk out of bounds in g
                then
                    continue;
                end
                if g[xi, yj, zk] == f then
                    if !(visited.Contains(xi, yj, zk)) then
                        neighbours.Enqueue(xi, yj, zk);
                        visited.Add(xi, yj, zk);
                    end
                end
            end
        end
    end
end
```

Algorithm 3: Flooding

3.4.5 Progress

All of the code for this stage of the pipeline was written from scratch. Most time was spent optimising for both time and space complexity once the algorithms used had been implemented.

Debugging was done by partitioning the voxel structure of a cube and colouring the voxels according to their fragments. The generating points were also visualised. This is shown in Figure 3.8.

3.4.6 Advantages

As discussed, while the partitioning here is done using a Voronoi method which is common when working with surface representations, having a volumetric representation of an object could allow more complex and physically accurate algorithms. For example, the grid of voxels could be taken to represent the crystalline structure of some solids with the 'bonds' between neighbouring voxels acting as they would in a real collision.

Also, the method works on both convex and concave models equally which is uncommon in other approaches as discussed in Section 1.2.

Figure 3.8: A visualisation of the Voronoi partitioning of the voxel structure of a cube.

3.5 Intermediate Processing

Before the partitioned voxels can become rigid meshes, preprocessing is needed. This is in order to prepare the data for the different algorithms used.

3.5.1 Separating Border Voxels

For the chosen meshing processes, the voxel grid is trimmed so that only interior border voxels remain, as shown in Figure 3.9. The reason for this is that both meshing processes require only the surface of the fragments defined by the voxels and not the solid volume. Separation is done by checking for every voxel whether it has a neighbouring voxel with a different colouring and keeping only those that do.

Voxels on the surface of the object, with a non-existent neighbour, are recorded for entry into a k-d tree.

A k-d tree is a binary tree based data structure which features an average $\mathcal{O}(n \log n)$ lookup for the nearest vector in the tree to a given vector in space.

An implementation of k-d trees in C# for Unity3D was found and used as there was little point reimplementing a well used data structure [1].

Voxels both on the surface and on the interior borders with other fragments are also stored in the fragment they belong to.

Figure 3.9: A fragmented voxelised rectangle. The interior border voxels are green and the exterior voxels are blue. The unneeded interior voxels are red.

3.6 Mesh Generation

Once the solid volume for each fragment has been found, it needs to be transformed into a rigid mesh for use in the physics engine. These meshes are constructed via two processes. The first splits the original object mesh and the second forms new meshes around the solid voxel structures.

3.6.1 Data Structures

The mesh partitioning procedure takes as an input the k-d tree of surface voxels from the intermediate processing stage as well as the original object mesh. The output is a list of fragment submeshes and for each one, a k-d tree of the vertices which make up the border of that submesh. These meshes will be referred to as the external meshes. This is shown in Figure 3.10.

The marching tetrahedra algorithm runs for each fragment and takes as an input the grid of coloured border voxels from the intermediate processing stage as well as the k-d tree of the vertices which make up the fragment's external mesh border as outputted by the mesh partitioning procedure above. The algorithm outputs a mesh which represent the surface defined by the voxels (referred to as the solid mesh), and surface mesh, for that fragment.

3.6.2 Mesh Partitioning

In order to preserve the original surface of the object the correct sections of the original mesh need to be mapped onto the computed fragments.

To do this the original object mesh is partitioned according to the colouring of the nearest voxel for every vertex.

This is the purpose of having all of the surface voxels in a k-d tree. There is no need to have the internal voxels as there will always be a surface voxel which is closer.

The process loops over all triangles and each vertex in that triangle. If all of the vertices have a nearest voxel of the same colouring then that triangle is added to the mesh for the fragment of that colouring. If the nearest voxel colours differ then the triangle is discarded and the vertices are added to the boundary k-d tree for their fragments.

The result of this process is demonstrated in Figure 3.10.

Figure 3.10: A partitioning of the Stanford Bunny mesh. The vertices which make up the border of the green submesh are highlighted in red. The gap between partitions is caused by the loss of triangles which have vertices in different partitions.

3.6.3 Marching Tetrahedra

Marching Cubes

Marching cubes is an algorithm used to extract a polygonal mesh of an implicit surface from a three-dimensional discrete scalar field [11]. The binary field of voxels[6] representing the solid volume of an object in this case is such an example.

The algorithm proceeds through the scalar field, taking a 'cube' of eight adjacent locations at a time. The polygon(s) needed to represent the part of the implicit surface that passes through this cube are determined by creating an index into a pre-calculated array of 256 configurations from the eight scalar values.

Ordinarily each vertex of the generated polygons would be placed on the appropriate position along the cube's edge by linearly interpolating the two scalar values that are connected by that edge. However, in this case the scalar values only have two possibilities and so interpolating between them always results in a placement of the vertices half way between the voxels. This leads to a meshed surface which reveals the blocky voxel structure as opposed to a smoother surface which would be extracted if dealing with a field of actual scalars. This is shown in Figures 3.11 and 3.12.

Ambiguous cases are present in the polygonisation where there exist at least two correct interpretations of the implicit surface. This is shown in two dimensions[7] in Figure 3.13.

[6]The colouring of the voxels is for identifying which fragment they belong to, they are still treated as binary present or not present for meshing purposes.

[7]These ambiguities would therefore be for the marching squares algorithm but the premise is the same.

Figure 3.11: Marching tetrahedra applied to a scalar field generated from perlin noise [17].

Figure 3.12: Marching tetrahedra applied to the voxel grid of a fragmented Stanford Bunny.

Figure 3.13: The two possible surfaces for the imaginary square shown with dashed lines with two points inside the surface and two outside.

Marching Tetrahedra

Marching tetrahedra is an algorithm based on marching cubes which solves these ambiguities as well as producing a surface which matches the voxel data more closely at the cost of generating more vertices [3].

In marching tetrahedra, each cube is split into six irregular tetrahedra by cutting the cube in half diagonally three times as shown in Figure 3.14. These tetrahedra are then used to find the implicit surface in much the same way as marching cubes however there are now nineteen edge intersections per cube instead of twelve.

Figure 3.14: A cube split into six irregular tetrahedra [8].

Modification

As two separate meshes are being produced in this stage, a section of the original mesh and a surface representing the solid volume, the two need to be combined and joined.

The marching tetrahedra library used was modified to take the grid of coloured border voxels as well as the set of voxels lying both on the exterior and on the fragment border. The k-d tree of the vertices which make up the fragment's external mesh border is also used. These border voxels are meshed using marching tetrahedra. However, when a voxel also lying on the exterior is found, the calculated vertices are discarded and instead their nearest neighbours in the k-d tree are used. This process knits the gap that would have otherwise existed between the two generated meshes, the external mesh and the solid mesh. A small amount of random noise is also applied to the rest of the vertices in an effort to disguise some of the blocky underlying structure. The results of this can be seen in Figure 3.15.

3.6.4 Progress

This stage of the pipeline took the longest to complete and was the final milestone when done. The stage went through several iterations of varying success before the above method was devised.

Originally the fragment meshes were to be only generated using marching tetrahedra with no splitting or use of the original external mesh. This resulted in an important detail loss as seen in Figure 3.12 and was not viable for use. Increasing the voxel resolution improved this but added too much of a computational overhead.

A second method involved only splitting the original external mesh and not using marching tetrahedra at all. The resultant fragment meshes were duplicated and had their normals flipped before being combined. This gave results that resembled hollow porcelain objects as seen in Figure 3.16. This could not be used as it did not satisfy the requirement of modelling the internal structure.

Producing the final method, involved modifying the marching cubes library found to stitch the external and solid meshes together as discussed. The algorithm found in the library was also optimised. Originally the algorithm would produce duplicate vertices for triangles which should share a vertex. This was solved by using a dictionary of vertices and indices into the vertex list. All vertices required by a triangle are checked in the dictionary, if they exist as a key then the index stored is used else the vertex is added to the vertex list and a new index stored in the dictionary with that vertex and used in the triangle. This dramatically reduced vertex counts in the meshes produced which is necessary as Unity3D imposes a limit of 65,534 vertices for any mesh.

Figure 3.15: The Stanford Bunny after the entire pipeline has completed.

Figure 3.16: The Stanford Bunny fractured as if it were a hollow object.

3.7 Post-processing

After the fragment meshes have been generated, some post processing must take place in order for the meshes to be used in the physics simulation.

3.7.1 Physics

One advantage of fragmenting a voxel structure is that the mass of each fragment, m_f, can be calculated using Equation 3.4.

$$m_f = m_o \times v_f / v_o \quad v \text{ is the number of voxels} \tag{3.4}$$

3.7.2 Game-object Generation

Finally, the fragment meshes and masses are passed to Unity3D so that they can be instantiated into the simulated world. Depending on the number of vertices in each mesh this can add a large time overhead which will be discussed in Section 4.2.2. This overhead is a limitation of Unity3D and not Dynamic Volumetric Fragmentation.

3.8 Result

The result is a library which can be used to fragment objects on collision based on the physical attributes of the object and of the collision.

The library produced requires that two scripts be attached to the destructible object, one as a driver for the destruction process and the other to allow the physical properties of the object to be defined. A Unity3D object with a 'sphere collider' must also be attached with a script which triggers the destruction process as detailed in Section 3.2. All of these elements are added to an existing object and none require any existing features of the object to be changed.

When an object is given the above attributes and a projectile fired at it, the object is broken into fragments and these fragments then receive force from the projectile causing them to explode out as would be expected. This is shown in Figure 3.17.

A video demonstrating the results can be found at `https://www.screenr.com/NRQN`.

The nature of the fragmentation differs depending on the forces involved as shown in Figure 3.18.

Figure 3.17: Fragments exploding out of the Stanford Bunny after a cubic projectile, shown in red, has been fired at it.

Figure 3.18: The biggest fragments after three Stanford Bunnies have been fragmented. Compared to the middle, the right was given a higher strength while the left was hit with a faster projectile.

3.9 Conclusion

In order for the proposed solution to be realised, four stages of implementation had to take place:

1. Preprocessing
2. Voxelisation
3. Meshing
4. Postprocessing

Intermediate stages were also completed to process data passed between the above sections.

Points 2 and 3 required adapting existing implementations of the algorithms used whereas the remaining sections were developed from scratch.

Chapter 4

Evaluation

This chapter will investigate the success of Dynamic Volumetric Fragmentation as a solution for computing the destructive results of physics-based collisions. Visual evaluation of the results produced will be carried out as well as empirical comparison with another destruction library for Unity3D.

Further work which could be done to improve the solution as it is in its current state will also be discussed.

4.1 Comparison

The 'Meshinator' library available for Unity3D will be used for comparative evaluation [12]. This library also fractures objects dynamically based on impact force and user defined object strength, however the method differs from ours.

Meshinator performs fragmentation by intersecting a plane, positioned depending on the impact force and object strength, with the object and shearing along it. Triangles crossing the plane are clipped to it and the two resulting meshes are closed with a flat surface. This can be seen in Figure 4.1.

4.1.1 Advantages

Latency

As will be seen in Section 4.2, the Meshinator library has a lower latency between the start and end of the fracturing pipeline. The reasons for this will be explored in further detail.

Conservation of volume

Due to the mesh being partitioned exactly by a plane, none of the original mesh's volume is lost on fragmenting. This is not the case with Dynamic Volumetric Fragmentation for two reasons:

- Triangles from the original mesh are lost rather than clipped during mesh partitioning. See Section 3.6.2.

Figure 4.1: The Stanford Bunny fractured into two fragments by the 'Meshinator' library.

- When applying marching tetrahedra to two adjacent volumes, there will be a gap between the two resulting meshes.

With our method, volume loss is minimised with increased triangle density in the original mesh as well as voxel resolution, at a computational cost. This is shown in Figures 4.2 and 4.3.

4.1.2 Disadvantages

The following are some limitations of the Meshinator library compared to Dynamic Volumetric Fragmentation.

Concavities

As discussed in Section 1.2, Meshinator cannot correctly fracture objects with concave sections. This is because of how the shearing is implemented. All of the vertices created by clipping triangles to the plane are added to new triangles in pairs with the third vertex being a point at the centre of the bounding box around these vertices. The result of this is shown in Figures 4.4 and 4.5. DVF does not have this limitation for the reasons discussed in Section 1.3.

Figure 4.2: The sphere has a low triangle count and so when the original mesh is partitioned and triangles are lost, a larger volume is lost in the end result.

Figure 4.3: The Stanford Bunny has a higher triangle count and so less volume is lost in the end result.

Figure 4.4: The incorrect fragments generated when Meshinator shears across a concavity.

Figure 4.5: The incorrect fragments generated when Meshinator shears across a concavity. Seen from another angle.

Accuracy

The visual accuracy of the Meshinator library is limited as it is only able to create flat planar shears when fragmenting. This is only physically accurate for some materials such as crystal. DVF allows for the partitioning pipeline stage to be easily swapped depending on the desired material properties, with the presence of the volume data giving a high number of possibilities.

Also, the Meshinator library always produces the same shear under the same impact, which is not realistic. Due to the distribution of the Voronoi generating points, DVF gives varied results under the same forces. This gives the realistic impression that no two objects are truly exactly the same. A comparison can be seen in Figures 4.6 and 4.7.

Figure 4.6: When two identical objects are fragmented under the same forces using Meshinator the results are also identical.

Figure 4.7: When two identical objects are fragmented under the same forces using this solution the results vary slightly as they would in the real world.

4.2 Empirical Evaluation

The main focus of this project was to investigate a solution which enables physically accurate object fragmentation. However, testing the physical accuracy implemented by the current partitioning pipeline stage empirically is beyond the scope of the project. Limitations exist in the Voronoi based partitioning as well as in the meshing stage with regards to physical accuracy. These problems, as well as whether the project is a viable starting point for more accurate fragmentation algorithms to be used in will be discussed in Section 4.4.1.

Instead, comparative empirical evaluation will be done between the Meshinator library and this implementation regarding use as a real-time solution for physical destruction.

4.2.1 Metrics

The following metrics are used for evaluation.

Latency

The delta time from the start of the fragmentation process to the end. In the case of Dynamic Volumetric Fragmentation this has been split for each pipeline stage.

This is the most important metric as the destructible object cannot visually fragment until the fragmentation stage is complete. As this metric increases so does the visible stutter between collision[1] and the object fragmenting. As it is the case that both solutions run on the main Unity3D physics thread, the entire scene will also pause for this time. This is discussed further in Section 4.4.2.

[1]Technically pre-collision in DVF. See Section 3.2.

Framerate

The is the number of frames being displayed per second.

This is important for a real-time application as it shows how smooth viewing the scene is. As it decreases the stuttering between frames becomes visible to the human eye, which is deemed undesirable for most real-time applications.

Frametime

This is the inverse of framerate. Frametime is a measure of how long it takes each frame to be processed.

Arguably, when it comes to evaluating performance, frametime is a better measure than framerate. This is because framerate is not linear in the amount of work done per frame, whereas frametime is.

4.2.2 Unity3D Overhead

One of the largest causes of latency in both solutions is caused by Unity3D, making it unavoidable when this engine is used as a base.

When instantiating new dynamically defined objects into the game world, depending on the complexity of the object meshes, a latency overhead is introduced. This latency cannot be avoided by multi-threading as the instantiation always takes place on the main thread.

Fortunately, as this is common to all implementations and is measurable it can be ignored[2].

To show that this is the case, a scene was constructed where, on collision, the Stanford Bunny was replaced with fragments with precomputed meshes identical to those produced from a run of Dynamic Volumetric Fragmentation.

The average time taken to instantiate 22 precomputed fragments was 1.23 seconds. Figure A.1 shows the full results over 10 runs.

[2]Though it will vary depending as the complexity and number of the produced fragments does.

4.2.3 Simple Object

Both methods were tested with a simple scene in which a sphere consisting of 760 triangles underwent a single collision.

Latency

Meshinator has a negligible latency when fracturing an object with a low triangle count.

DVF has a much higher, noticeable latency when fracturing an object with a low triangle count. This is because the complexity is also bounded to the voxel resolution as well as triangle density. This is confirmed by the fact that the highest latency pipeline stages are the voxel partitioning and marching tetrahedra.

The standard deviation for the mean total time for DVF is higher than for Meshinator. This is because Meshinator always produces identical fragments each time[3] whereas the proposed solution produces different fragments as well as a varying number of fragments.

	Meshinator	DVF
Instantiation	0.00532	0.56589
Marching Tetrahedra		3.19783
Mesh Partitioning		0.02593
Voxel Partitioning		0.19958
Voxelisation		0.00524
Fracturing	0.03290	

Figure 4.8: The mean time taken by each pipeline stage in seconds when fracturing a sphere consisting of 760 triangles over 10 runs. Figures A.2 and A.7 show the full tables.

[3]See Section 4.1.2

Framerate

Both solutions have an excellent framerate of 120 frames per second before and after the collision.

Meshinator does not have a complete framerate drop to 0 whereas DVF does. This is likely to be because Meshinator has multi-threaded components whereas DVF does not. Also, the instantiation overhead will be lower for Meshinator as it only created two fragments, DVF created 22.

Overall, Meshinator spends less time below 120 frames per second.

Figure 4.9: The framerate when fracturing a sphere consisting of 760 triangles. The red line indicates the collision time.

Frametime

As would be expected, in both cases there is a frametime spike after the collision as more work has to be done per frame as the object is fragmented.

DVF has a longer period between the collision and frametime spike and there are no frames produced at all before the pipeline is complete.

Figure 4.10: The frametime when fracturing a sphere consisting of 760 triangles. The red line indicates the collision time.

4.2.4 Complex Object

Both methods were tested with a simple scene in which the Stanford Bunny, consisting of 69,666 triangles, underwent a single collision.

Latency

When fragmenting a higher triangle density object, Meshinator begins to have a noticeable latency, however, this is likely due to the instantiation overhead. Though an increase in latency would be expected with more complex objects as the complexity is bound to the number of triangles.

As expected, DVF has a much higher latency when fragmenting complex objects. The mesh partitioning stage now attributes to much more of this latency, with the voxel partitioning stage only having a slight increase. This is because the former is bound to the number of triangles whereas the latter is bound to voxel resolution, which has not increased much compared to the simple sphere. The instantiation phase now has a much higher standard deviation as the triangle density of the outer fragments (those that contain portions of the original mesh) is much higher than in the sphere case. The triangle density of the inner fragments is relatively unchanged as this depends on the voxel resolution.

Framerate

The framerate over time progresses in much the same way as with the simple object, however in both cases the period before regaining 120 frames per second is longer.

Meshinator now has a full drop to 0 frames per second. This is likely due to the instantiation overhead as the two fragments being created are of higher triangle complexity.

Frametime

Frametime is as it was with the simple object, however the period between the collision and the spike is longer as would be expected.

Figure 4.11: The mean time taken by each pipeline stage in seconds when fracturing the Stanford Bunny, consisting of 69,666 triangles, over 10 runs. Figures A.3 and A.6 show the full tables.

Figure 4.12: The framerate when fracturing the Stanford Bunny, consisting of 69,666 triangles. The red line indicates the collision time.

Figure 4.13: The frametime when fracturing the Stanford Bunny, consisting of 69,666 triangles. The red line indicates the collision time.

4.2.5 Multiple Objects

Both methods were tested with a simple scene in which two Stanford Bunnies each underwent a single collision simultaneously.

Latency

In both cases the latency has roughly doubled as the number of objects has.

The standard deviation for the total time has not increased significantly for Meshinator, again because the fragments are always identical. For DVF it has increased as would be expected.

Framerate

Framerate follows the same trend as before in both cases, with the period where no frames are being produced having increased as mean total latency did.

Frametime

Frametime is as with the previous two test cases, with the period before the frametime spike increasing as the number of objects has.

It is still important to note that, though the frametime is 0, there are still no frames being produced between the collision and the spike, as shown by the framerate graph.

Figure 4.14: The time taken in seconds to simultaneously fracture two Stanford Bunnies, each consisting of 69,666 triangles, over 10 runs. Figures A.4 and A.5 show the full tables.

Figure 4.15: The framerate when simultaneously fracturing two Stanford Bunnies, each consisting of 69,666 triangles. The red line indicates the collision time.

Figure 4.16: The frametime when simultaneously fracturing two Stanford Bunnies, each consisting of 69,666 triangles. The red line indicates the collision time.

4.2.6 Results

It is clear from these results that Dynamic Volumetric Fragmentation is not feasible for a real-time application in its current state. Even if the instantiation overhead were not present, there is too much latency in the other pipeline stages.

The Meshinator library would also not be usable in a realtime situation based on the results seen in the complex and multi-object trials. However a much greater proportion of the observed latency is due to instantiation than is the case for DVF. Furthermore, Müller, Chentanez and Tae-Yong demonstrate that real-time fragmentation can be achieved even if Meshinator is not a suitable example [13].

Meshinator likely outperformed DVF in these empirical tests for the following reasons:

1. The plane intersection method is only bounded in time complexity by the number of triangles. DVF is also bounded by voxel density which is independent of object triangle complexity.
2. Meshinator only produces two fragments per collision which reduces the work done to produce the fragment meshes as well as instantiate them. DVF produces a variable number of fragments.
3. Meshinator produces fragments with shear surfaces reducing the triangle complexity of the fragments. DVF creates noisy surfaces.
4. Meshinator is fully multi-threaded. DVF is not.

Whilst Points 2 and 3 reduce the time complexity of Meshinator, they also reduce its functionality. DVF is much less strict in what it is capable of producing and so can achieve more accurate and varied fragmentations which depend better on the physical attributes of the solid.

Point 4 will be addressed in Section 4.4.2.

4.3 Requirements Met

The requirements, originally stated in Section 2.4, are enumerated below alongside evidence of their completion:

1. The solution *must* enable a scene in which an object of reasonable complexity[4] is fragmented in a manner which resembles a real world interaction as closely as possible. Demonstrated in Section 3.8
2. The solution *must* be object independent. That is any object may be made destructible without retroactively altering the object's definition. Demonstrated in Section 3.8
3. The solution *must* take into account the defined physical characteristics of the object to be fragmented as well as those defined by the collision when resolving. Demonstrated in Sections 3.2 and 3.4

[4]Such as the Stanford Bunny

4. The solution *should* produce fragments with the correct physical properties based on those of the original object and the size of the each given fragment. Demonstrated in Section 3.7.1

5. The solution *should* not produce fragments with unconnected islands. Demonstrated in Section 3.4.4

6. The solution *could* run in real-time.

7. The solution *won't* deal with texture mapping onto fragments.

8. The solution *won't* deal with structural integrity.

Point 6 unfortunately has not been achieved as stated in Section 4.2.6. Further work that could be done to address this is explored in Section 4.4.2.

As expected, neither Point 7 or 8 were achieved.

4.4 Feasible Continuations

Two areas for improvement are explored below.

4.4.1 Accuracy

The fragmentation method used, while producing visually pleasing results compared to Meshinator[5], is not as physically accurate as Dynamic Volumetric Fragmentation can allow.

Because the object is being modelled as a solid volume it would be possible to write much more complex algorithms which fully simulate how impact force should be propagated internally as well as how the cracks should form. This is the biggest strength of DVF.

As the pipeline stages are well encapsulated, it would also be possible to have different algorithms for different materials and use the correct one on an object by object basis.

4.4.2 Parallelism

In its current state, none of the pipeline is parallelised. There are two ways in which this could be improved.

Coarse-grained

Currently the entire pipeline runs in the same thread as the Unity3D physics engine which causes the whole scene to pause while the fragmentation resolves. Had time allowed, the pipeline would have been moved to its own thread to prevent this. Unity3D has many thread-unsafe areas and all of the stages were developed without such areas meaning that they could be used in a multi-threaded environment.

[5]Though the blocky structure is probably more visible than would be desired.

Only moving the whole pipeline to its own thread does not solve the latency problem however. Even if the stages are executing independently of the physics engine, allowing it to continue to progress, if they are not all complete between the time the collision is predicted[6] and the collision then the fragments will not interact with the colliding object as is required. Simply increasing the radius of collision prediction does not work as that will mean the assumptions made in Section 3.2 no longer hold.

Fine-grained

Within each pipeline stage, most operations are being performed on vertices or voxels independent of the results on the previous vertices or voxel. This would allow massive parallelism to be exploited on the GPU for these operations.

GPU parallelisation is the best option for allowing DVF to be viable for a real-time application. This would have been explored had time allowed.

4.5 Conclusion

The advantages and disadvantages of Dynamic Volumetric Fragmentation have been investigated with comparison to another solution. While it was found that DVF would not currently be suitable for a real-time application, possible continuations which would address this have been discussed, primarily involving parallelisation that could not be completed due to time constraints.

An argument has been made for the merits of using a volumetric approach to physics based destruction. It is suggested that the current partitioning algorithm could be replaced with one featuring higher physical accuracy which would improve the already pleasing visual results.

[6]See Section 3.2

Chapter 5

Conclusions

A functioning pipeline for dynamic physics-based fragmentation of objects based on their computed volumetric definition has been implemented, titled Dynamic Volumetric Fragmentation. This implementation can demonstrably be used with a physics engine to resolve post-collision fragmentation in a visually accurate manner, with further physical accuracy left as a possibility.

The problem of splitting an object precisely into fragments given only rigid mesh information has been solved by finding the entire enclosed volume as a scalar field at runtime. Partitioning of this scalar field algorithmically for each point in the field allows mass to be attributed to each fragment according to predefined rules inspired by real world physical principles.

While this implementation of DVF is not viable for real-time applications, a possible plan for achieving real-time suitability has also been outlined, focussing on parallelism.

DVF could be used in the field of Computer-Generated Imagery, where frame render times are generally much higher and do not have to be viewed in real-time. This would allow objects to be accurately fragmented without the need for an artist's input.

Continuations which would further improve the viability of DVF have been discussed in Section 4.4.

Bibliography

[1] Andeeeee. Point nearest neighbour search class. http://forum.unity3d.com/threads/point-nearest-neighbour-search-class.29923/. Last Accessed: 2015-04-12.

[2] Alex Benton. Ray tracing - all the maths. Advanced Graphics Lecture, 2015.

[3] Paul Bourke. Polygonising a scalar field using tetrahedrons. http://paulbourke.net/geometry/polygonise/. Last Accessed: 2015-04-12.

[4] Miguel Cepero. Appetite for destruction. http://procworld.blogspot.co.uk/2014/08/appetite-for-destruction.html. Last Accessed: 2015-04-12.

[5] Miguel Cepero. Procedural world. http://procworld.blogspot.co.uk/. Last Accessed: 2015-04-12.

[6] Vincenzo Giovanni Comito. Voxelization in unity. http://nihilistdev.blogspot.co.uk/2012/08/voxelization-in-unity.html. Last Accessed: 2015-04-12.

[7] Wikimedia Commons. Euclidean voronoi diagram. http://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg. Last Accessed: 2015-04-12.

[8] Wikimedia Commons. Marching tetrahedrons. http://commons.wikimedia.org/wiki/File:Marching_tetrahedrons.png. Last Accessed: 2015-04-12.

[9] Wolfgang Engel. *GPU PRO 3: Advanced Rendering Techniques*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition, 2012.

[10] Sarah F. Frisken Gibson. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In *Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, MICCAI '98, pages 888–898, London, UK, UK, 1998. Springer-Verlag.

[11] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[12] MD. Meshinator. http://meshinator.blogspot.co.uk/. Last Accessed: 2015-04-12.

[13] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, 32(4):115:1–115:10, July 2013.

[14] Matthias Nießner, Christian Siegl, Henry Schäfer, and Charles Loop. Real-time collision detection for dynamic hardware tessellated objects. 2013.

[15] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics*, 29(6 (Proceedings of SIGGRAPH Asia 2010)):179:1–179:9, December 2010.

[16] Scrawk. Marching cubes plugin for unity. `http://scrawkblog.com/2013/04/01/marching-cubes-plugin-for-unity/`. Last Accessed: 2015-04-12.

[17] Scrawk. marchingtetrahedrons.jpg. `http://scrawkblog.com/2014/06/24/directcompute-tutorial-for-unity-introduction/`. Last Accessed: 2015-04-12.

Appendix A

Expanded Tables

Latency (s)										Mean	Std. Deviation
1.22708	1.22354	1.23008	1.22643	1.22024	1.22292	1.22583	1.23023	1.22098	1.22292	1.22502	0.00331

Figure A.1: The time taken in seconds to instantiate 22 precomputed fragments over 10 runs.

	Latency (s)										Mean	Std. Deviation
Fracturing	0.03295	0.03298	0.03218	0.03218	0.03298	0.03340	0.03357	0.03274	0.03306	0.03299	0.03290	0.00042
Instantiation	0.00614	0.00521	0.00522	0.00566	0.00523	0.00505	0.00585	0.00507	0.00498	0.00482	0.00532	0.00040
Total	0.03908	0.03819	0.03740	0.03784	0.03821	0.03845	0.03941	0.03782	0.03804	0.03782	0.03823	0.00058

Figure A.2: The time in seconds when fracturing a sphere consisting of 760 triangles over 10 runs using Meshinator.

	Latency (s)										Mean	Std. Deviation
Fracturing	0.18868	0.18612	0.18912	0.18824	0.18727	0.18748	0.18771	0.18653	0.18930	0.18642	0.18769	0.00108
Instantiation	0.84425	0.81771	0.82560	0.83062	0.82873	0.82464	0.83161	0.82826	0.82626	0.83006	0.82878	0.00639
Total	1.03294	1.00383	1.01472	1.01885	1.01600	1.01212	1.01932	1.01479	1.01557	1.01648	1.01646	0.00686

Figure A.3: The time in seconds when fracturing the Stanford Bunny, consisting of 69,666 triangles,s over 10 runs using Meshinator.

Latency (s)										Mean	Std. Deviation
1.88355	1.86621	1.86596	1.86734	1.83400	1.85642	1.87400	1.86322	1.85852	1.85900	1.86282	0.01225

Figure A.4: The time taken in seconds to simultaneously fracture two Stanford Bunnies, each consisting of 69,666 triangles, over 10 runs using Meshinator.

Latency (s)										Mean	Std. Deviation
21.54632	22.42930	23.07711	23.05247	24.14008	24.31929	21.50906	23.39180	24.14957	21.81323	22.94282	1.02769

Figure A.5: The time taken in seconds to simultaneously fracture two Stanford Bunnies, each consisting of 69,666 triangles, over 10 runs using DVF.

	Latency (s)										Mean	Std. Deviation
Voxelisation	0.02680	0.02643	0.03758	0.02549	0.02027	0.02192	0.02927	0.02313	0.01642	0.03108	0.02584	0.00567
Voxel Partitioning	0.65588	0.65310	0.65346	0.65739	0.65460	0.64515	0.66083	0.64542	0.65323	0.65762	0.65367	0.00477
Mesh Partitioning	1.41851	1.43271	1.42792	1.42263	1.43267	1.42783	1.42622	1.42306	1.42512	1.44006	1.42767	0.00586
	0.59194	1.01915	0.58903	0.54119	0.71245	0.46984	0.70034	0.54277	0.69631	0.56030		
	0.15911	0.18107	0.18746	0.07855	0.10072	0.16924	0.15909	0.08264	0.09236	0.09493		
	0.12167	0.17524	0.20578	0.15686	0.09406	0.06987	0.07093	0.09298	0.22174	0.07177		
	0.24118	0.22795	0.36229	0.11783	0.15856	0.38301	0.11796	0.32991	0.12409	0.12925		
	0.65095	0.19210	0.05928	0.21263	0.10330	0.24357	0.08137	0.07098	0.64050	0.23812		
	0.19863	0.41837	0.38166	0.17141	0.29239	0.16990	0.21482	0.26338	1.12057	0.15758		
	0.26686	0.66692	0.11701	0.33353	0.16193	0.20947	0.39053	0.04507	0.36556	0.15342		
	0.18016	0.18585	0.13598	0.50836	0.11880	0.14050	0.22221	0.22572	0.10073	0.34041		
	0.08645	0.31989	0.04849	0.26240	0.10213	0.43531	0.41169	0.32231	0.14262	0.31997		
	0.23820	0.13864	0.42792	0.47167	0.16955	0.10200	0.29265	0.17137	0.19248	0.33092		
	0.10417	0.19069	0.08262	0.17472	0.31186	0.12411	0.14823	0.06499	0.09163	0.29213		
	0.57624	0.36273	0.20114	0.14634	0.04445	0.22288	0.28074	0.12165	0.16330	0.10976		
	0.06883	0.28798	0.05984	0.21202	0.45770	0.20855	0.43295	0.20529	0.10564	0.08113		
	0.31951	0.29608	0.07177	0.25367	0.21801	0.26132	0.12386	0.38551	0.40807	0.08003		
	0.14779	0.09003	0.29298	0.25725	0.37468	0.20625	0.19784	0.11512	0.29583	0.12765		
	0.05370	0.04378	0.02869	0.07413	0.17734	0.18861	0.12482	0.22509	0.43806	0.19690		
	0.06191	0.46145	0.40708	0.56472	0.29336	0.12935	0.02794	0.18961	0.18952	0.23416		
	0.13339	0.08915	0.06830	0.18124	0.15111	0.41745	0.23373	0.48318	0.53899	0.08974		
	0.68372	0.67057	0.04953	0.38026	0.20501	0.35432	0.29522	0.19699	0.27676	0.25860		
	0.10499	0.13849	0.34652	0.24293	0.36568	0.17025	0.27992	1.18381	0.11161	0.21334		
	0.29401	0.30109	0.34935	0.43989	0.42993	0.39082	0.12265	0.50032	0.23934	0.41442		
	0.29013	0.24914	0.12950	0.61354	0.21097	0.21578	0.09043	0.23130	0.34831	0.01794		
	0.16420	0.33848	0.23214	0.32996	0.20722	0.12243	0.14022	0.12938	0.17640	0.36979		
	0.04162	0.65245	0.26871		0.30292		0.02862			0.58460		
Marching Tetrahedra			0.06031				0.19266				0.25480	0.18492
All Fragments	5.77936	7.69729	5.16337	6.72512	5.76415	5.40482	5.38141	6.17941	7.08044	5.46686	6.06422	0.79939
	1.62070	1.60895	1.65004	1.55500	1.73827	1.70221	1.61766	1.67268	1.58440	1.75882		
	0.20854	0.01326	0.10191	0.01351	0.15348	0.08233	0.08321	0.04516	0.04527	0.08252		
	0.02944	0.05321	0.02939	0.02454	0.02770	0.03653	0.03998	0.01684	0.01697	0.03980		
	0.02062	0.01785	0.10727	0.01446	0.02075	0.01682	0.02251	0.02191	0.02928	0.02800		
	0.02827	0.04366	0.03494	0.00589	0.02210	0.02621	0.00615	0.03073	0.01965	0.06816		
	0.08210	0.01927	0.02143	0.02362	0.01945	0.02057	0.05721	0.03988	0.04220	0.01329		
	0.00938	0.11551	0.03418	0.00464	0.03545	0.03964	0.07982	0.01011	0.05404	0.04125		
	0.07516	0.10964	0.02075	0.00783	0.02287	0.02579	0.01879	0.09096	0.02251	0.07117		
	0.03403	0.13915	0.00987	0.04081	0.01450	0.00743	0.01044	0.03291	0.01174	0.17284		
	0.03123	0.02548	0.00393	0.04079	0.05302	0.02660	0.18178	0.03954	0.02968	0.02858		
	0.00331	0.02516	0.11650	0.16368	0.19393	0.12717	0.04950	0.14776	0.10049	0.02195		
	0.02656	0.01924	0.10181	0.04767	0.01796	0.00437	0.01246	0.02658	0.14667	0.01334		
	0.02854	0.00120	0.02187	0.10701	0.02757	0.02572	0.03072	0.00138	0.01640	0.00771		
	0.02255	0.00789	0.00919	0.06089	0.00121	0.12547	0.01750	0.08959	0.01339	0.01241		
	0.00723	0.04971	0.03100	0.03646	0.02520	0.01449	0.01801	0.02807	0.01633	0.01484		
	0.00894	0.01233	0.00188	0.03174	0.01273	0.01313	0.00814	0.02887	0.05573	0.00638		
	0.00542	0.00589	0.02115	0.00393	0.01535	0.02482	0.02921	0.01872	0.01225	0.03415		
	0.00577	0.00492	0.01251	0.03578	0.00612	0.02989	0.01412	0.01278	0.01624	0.00647		
	0.00332	0.01601	0.01091	0.02956	0.00198	0.00700	0.01014	0.00558	0.01335	0.01760		
	0.00710	0.00572	0.00279	0.02096	0.02771	0.01483	0.01585	0.00636	0.00882	0.00060		
	0.01480	0.00516	0.00399	0.02346	0.01610	0.00326	0.00058	0.00147	0.01575	0.00217		
	0.00303	0.00425	0.00285	0.01855	0.00156	0.01290	0.01742	0.01479	0.01864	0.00913		
	0.02069	0.01530	0.02074	0.01123	0.01323	0.00877	0.01138	0.01624	0.03425	0.00387		
	0.00388	0.01634	0.00102		0.00651		0.00086			0.01124		
Instantiation			0.00289				0.00082				0.09978	0.32720
All Fragments	2.30061	2.33510	2.37479	2.32202	2.47474	2.39596	2.35425	2.39891	2.32407	2.46629	2.37467	0.05679
Total	**10.18115**	**12.14463**	**9.65712**	**11.15264**	**10.34642**	**9.89569**	**9.85198**	**10.66993**	**11.49926**	**10.06190**	**10.54607**	**0.77134**

Figure A.6: The time in seconds by each pipeline stage when fracturing the Stanford Bunny, consisting of 69,666 triangles, over 10 runs using DVF. The number of generated fragments can vary slightly, resulting in the number of times the marching tetrahedra and instantiation phases are applied differing between runs.

	Latency (s)										Mean	Std. Deviation
Voxelisation	0.00456	0.00500	0.00509	0.00507	0.00569	0.00519	0.00508	0.00543	0.00587	0.00543	0.00524	0.00036
Voxel Partitioning	0.19798	0.20167	0.19834	0.19666	0.20683	0.19849	0.19552	0.20255	0.20284	0.19491	0.19958	0.00359
Mesh Partitioning	0.02882	0.02434	0.02567	0.02619	0.02751	0.02466	0.02627	0.02667	0.02461	0.02458	0.02593	0.00139
	0.81361	0.92120	0.84699	0.93016	0.75813	1.03800	0.78382	0.75576	0.76424	0.86447		
	0.09411	0.28422	0.15782	0.12915	0.06455	0.00825	0.08385	0.09040	0.20059	0.06471		
	0.06664	0.04095	0.06691	0.04672	0.06925	0.07337	0.08310	0.05806	0.18593	0.09959		
	0.09389	0.08019	0.06640	0.09066	0.13722	0.08692	0.07341	0.04900	0.17204	0.05972		
	0.05500	0.13825	0.03910	0.11636	0.10829	0.11044	0.12475	0.14003	0.13219	0.14740		
	0.15684	0.06148	0.09853	0.14906	0.04698	0.09650	0.05647	0.14476	0.14456	0.11246		
	0.05842	0.13193	0.08106	0.06578	0.22873	0.08380	0.07199	0.26092	0.21343	0.11746		
	0.07579	0.17009	0.07680	0.17620	0.22062	0.05704	0.11178	0.18724	0.08183	0.06393		
	0.05610	0.08768	0.08374	0.16799	0.03988	0.12498	0.06808	0.12637	0.12948	0.07968		
	0.15411	0.16552	0.10199	0.10153	0.03528	0.13434	0.22246	0.09386	0.10457	0.06881		
	0.07280	0.13340	0.19631	0.05363	0.10641	0.08416	0.18046	0.07492	0.17328	0.12114		
	0.21608	0.03170	0.19632	0.07980	0.21498	0.09002	0.12570	0.09069	0.06699	0.11356		
	0.16441	0.04083	0.07262	0.08298	0.21378	0.10809	0.04881	0.19582	0.08522	0.14855		
	0.03850	0.28727	0.14180	0.08630	0.04295	0.14842	0.15885	0.24251	0.07279	0.18508		
	0.08089	0.10770	0.04048	0.04552	0.06279	0.19711	0.19464	0.04288	0.15105	0.17819		
	0.11183	0.07183	0.21447	0.04814	0.04337	0.14690	0.09987	0.08688	0.10577	0.19704		
	0.08072	0.03098	0.17789	0.10558	0.14161	0.12928	0.11160	0.03147	0.07691	0.11946		
	0.02694	0.08233	0.12413	0.08078	0.08662	0.06350	0.13643	0.06079	0.11908	0.07401		
	0.18085	0.13093	0.04728	0.33171	0.10562	0.29653	0.24445	0.21037	0.08303	0.10423		
	0.07846	0.06391	0.24567	0.20073	0.07923	0.18111	0.10379	0.20490	0.07823	0.20465		
	0.24405	0.19833	0.08768	0.17986	0.21597	0.07379	0.07909	0.10219		0.20355		
Marching Tetrahedra						0.12803					0.15228	0.16767
All Fragments	2.92004	3.26074	3.16399	3.26864	3.02225	3.46056	3.16338	3.24982	3.14120	3.32768	3.19783	0.14529
	0.10828	0.10482	0.08121	0.07954	0.09138	0.09515	0.10539	0.09222	0.08796	0.06423		
	0.06129	0.09180	0.02655	0.06488	0.04369	0.10036	0.02579	0.02645	0.03806	0.02372		
	0.06017	0.03659	0.09653	0.01804	0.02613	0.03430	0.03409	0.07078	0.02857	0.04389		
	0.01780	0.01456	0.00982	0.01192	0.00517	0.00996	0.03935	0.04104	0.02035	0.06022		
	0.04617	0.01548	0.10955	0.10202	0.03512	0.03193	0.03794	0.02029	0.04018	0.01528		
	0.00361	0.10307	0.00928	0.02118	0.00627	0.01087	0.03623	0.00734	0.02957	0.01810		
	0.00723	0.01068	0.02983	0.00213	0.09956	0.02631	0.01107	0.02160	0.01073	0.00910		
	0.00757	0.02081	0.00998	0.02419	0.00761	0.00420	0.01435	0.01742	0.03332	0.01200		
	0.00886	0.00417	0.00677	0.04549	0.00437	0.01106	0.02038	0.01141	0.00378	0.02790		
	0.01343	0.07631	0.00523	0.00178	0.00482	0.00542	0.00376	0.01578	0.01161	0.00794		
	0.00742	0.01544	0.02427	0.01400	0.01436	0.01776	0.00276	0.04578	0.02054	0.00635		
	0.00725	0.01669	0.01602	0.01073	0.04334	0.01556	0.01737	0.01760	0.00610	0.11324		
	0.02325	0.00563	0.04417	0.01759	0.00739	0.02051	0.00593	0.01181	0.00429	0.01896		
	0.02185	0.00648	0.00546	0.01593	0.00168	0.01450	0.01267	0.01093	0.00487	0.00874		
	0.01807	0.01694	0.00246	0.03527	0.01685	0.00901	0.00505	0.00323	0.01902	0.03622		
	0.00176	0.01768	0.03000	0.01232	0.01437	0.00092	0.03359	0.01168	0.10898	0.01091		
	0.00243	0.00440	0.01782	0.00272	0.00872	0.01630	0.00785	0.01188	0.01496	0.01158		
	0.00620	0.00541	0.00357	0.00733	0.09458	0.02232	0.06231	0.00590	0.02703	0.00884		
	0.00994	0.01617	0.02017	0.02628	0.00834	0.03529	0.03719	0.03126	0.01823	0.03663		
	0.05514	0.00444	0.01564	0.00593	0.00683	0.01465	0.02949	0.11070	0.02526	0.05746		
	0.01999	0.01489	0.00567	0.02199	0.01933	0.01717	0.01577	0.00552		0.02037		
Instantiation						0.05003					0.02695	0.02768
All Fragments	0.50769	0.60244	0.57001	0.54126	0.55990	0.56358	0.55833	0.59061	0.55343	0.61169	0.56589	0.02883
Total	**3.65910**	**4.09419**	**3.96310**	**4.03781**	**3.82219**	**4.25248**	**3.94859**	**4.07509**	**3.92794**	**4.16429**	**3.99448**	**0.16230**

Figure A.7: The time in seconds by each pipeline stage when fracturing a sphere consisting of 760 triangles over 10 runs using DVF. The number of generated fragments can vary slightly, resulting in the number of times the marching tetrahedra and instantiation phases are applied differing between runs.