Encoding Functionality Directly into Polygonal Models

Ben Lee[1] and Stephen Brooks[1][a]
[1]*Faculty of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, Canada*
{bn628547, sbrooks}@cs.dal.ca

Keywords: 3D Objects, Embedded Functionality, Meta Data

Abstract: Vertices in 3D meshes are typically stored with more precision than is needed for most use cases. We argue that this unused precision could be used to enhance the functionality of meshes by encoding additional metadata in the mesh itself. And with this approach, the user of the mesh can choose to utilize the embedded functionality without requiring any extra preprocessing steps, while those who ignore it can use the mesh normally. We apply this approach to two different use cases for the purpose of augmenting graphics functionality. We first show that we can store ambient occlusion to a level which is imperceptible to its vertex colored equivalent. We then use directly encoded information into the mesh to create a prefractured version of the mesh when subjected to impacts.

1 INTRODUCTION

Virtual scenes have increasingly large databases of 3D objects. Most commonly these objects are defined as polygonal meshes. These use a set of vertices as 3D coordinates in space, along with a list of indices into that vertex set describing how they are connected to create the polygons. But these objects typically remain static. Instead consider if meshes contained additional information that provides embedded functionality.

To achieve this, we can encode a small amount of metadata directly in the mesh that would later be used to enhance the mesh from its current state. To get a better idea of how to store this metadata, we can look into how polygonal meshes are formatted on disk. Meshes are made of a set of vertices described with three coordinates X, Y and Z, with each value stored with an IEEE formatted 32-bit floating point number.

In a floating-point number, each successive bit in the mantissa contributes less to determining the exact position of a vertex than the one before it. In practical terms, altering just a few of the least significant bits causes such a minimal change in position that it's virtually invisible to the human eye. The idea of this work is to store information in those unused bits that augments the functionality of the mesh.

The storage of this data is accomplished through operations that manipulate individual bits. Figure 1 shows an example where a single vertex with 3D coordinates $(0.2, 1.0, 0.5)$ is encoded with additional data with 3 bits per axis. The position changes so slightly that it is indistinguishable.

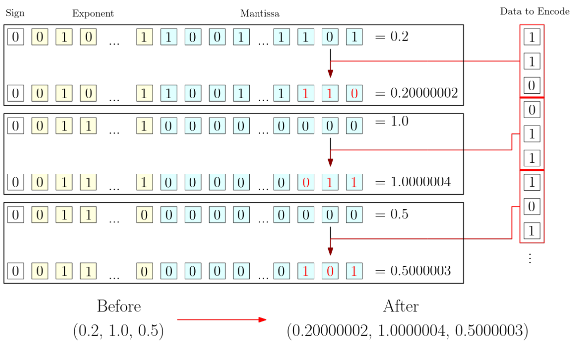


Figure 1: Example of encoding binary value 110011101 into a vertex's position (0.2, 1.0, 0.5) using three bits per axis.

Encoding metadata into the lower bits of the mantissa offers two key advantages. First, because the file's structure remains unchanged, the mesh can be used directly without any need for preprocessing. This makes it convenient for users, who can choose whether or not to utilize the embedded metadata. Second, there's no need for custom file formats, or dedicated fields in the file to store this information—only the vertex positions are required, which are inherently present in all 3D mesh formats.

This paper will begin with a discussion of related work. Then we analyze how many bits can be used

[a] https://orcid.org/0000000274735830

without causing visual artifacts in Section 3. We next apply this lower bit encoding for static ambient occlusion in 4. Following this, in Section 5 we store information in 3D meshes that allows a pre-fractured version of the mesh to be revealed.

2 PRIOR WORK

2.1 Ambient Occlusion

Ambient Occlusion (AO) techniques can generally be separated in two categories: static techniques and screen space techniques. Screen space methods are post processing effects applied after rendering, where each pixel can be used with its neighbours to determine how occluded it is. This typically involves sampling in a sphere or hemisphere around that pixel and determining how many of those samples are above and below the surface that pixel is a part of (McGuire et al., 2011; Mittring, 2007). The more samples above that surface generally means it is less occluded.

Our method is a static approach where AO is calculated ahead of time and saved inside the mesh to be used at runtime (Miller, 1994; Langer and Bülthoff, 2000). As an application of embedding functionality into the mesh, AO will be encoded directly into the vertices and applied with vertex shading.

2.2 Mesh Fracturing

There are several categories of mesh fracturing methods in computer graphics. Physically based methods are heavily researched. These use deterministic methods such as finite element methods (FEM) (O'Brien et al., 2002) and extended versions (XFEM) (Chitalu et al., 2020; Kaufmann et al., 2009) to simulate the stress and strain of a given impact across a mesh which then propagates cracks and splits the mesh into fragments. This is useful when highly realistic and convincing results are required, or when studying the properties of certain materials such as bone or concrete. Unsurprisingly, these methods are computationally expensive, and in most cases are not feasible in real time scenarios.

One method of real time fracturing is called prescoring, whereby a copy of the mesh is created, and an artist separates it into fragments manually through 3D modelling software such as Blender (Blender Development Team, 2022). Then at the time of impact these two meshes are instantly swapped to create the illusion of brittle fracture. The drawback is that there is only one version of a broken mesh, so it is less reactive to the type and location of impact.

Finally, there are methods for real-time fracturing, in which the fracture is reactive to user driven impacts. A popular method in these scenarios is fragmenting the object based on a voronoi pattern (Schvartzman and Otaduy, 2014). Although Voronoi based fracturing does not create realistic fracture patterns, it makes up for it in the fact that it is interactable (Müller et al., 2013). However, these approaches still typically assume a homogeneous material, and the added fracture pattern is still two dimensional, being applied only to the surface of the mesh. This makes them suitable for thin objects such as glass and walls, but they overall lack any 3D control.

Our method is based on prefracturing but metadata is stored in the lower bits and are used to reconstruct a set of cutting meshes. These separate the source mesh into different fragments, effectively providing a fractured version of the mesh at no extra storage cost. Sefakis et al. (2007) also make use of cutting meshes, however they make cuts along a tetrahedral volume in place of the original triangulated mesh which requires additional processing. They also do not embed the fracture information into the mesh vertices.

2.3 Compression and Watermarking

There are many methods that compress the storage requirements of the mesh. Deering (1995) notes that most of the space of a mesh is modeled within requires just the 24 bit mantissa, and the exponent typically stays the same. Deering uses a modified Huffman encoding to store the deltas between the vertices, essentially splitting the modeling space into a grid and storing the local position in that grid. Chou and Meng (2002) compress the vertices by mapping the input vectors to a codebook of codevectors, creating a lookup table for a fixed number of vectors.

However, we have different aims and want to be able to use the mesh without any additional processing, so we restrict ourselves to the encoding of metadata in the lower bits of the mantissa. In this regard, the storage requirement of the mesh will stay the same, but the bits inside of it will be more efficiently utilized.

There is also interest in asserting intellectual property rights over 3D meshes. One approach involves discreetly embedding a watermark within the mesh, which can later be retrieved to verify ownership (Wang et al., 2008). These watermarking techniques prioritize robustness—ensuring that the embedded data remains recoverable even if the mesh undergoes modifications. A notable example of such a method is presented by Benedens and Busch (2000). Cayre and Macq (2003) also introduced a method using a secret key to define a sequence of adjacent triangles within a mesh, which serves as a carrier for covert data. Yang et al. (2013) constrain the angle of surface normals during the encoding of uniform spatial noise.

3 How many bits?

We begin with the assumption that vertex positions are represented using 32-bit IEEE floating-point format, as it is the most widely adopted standard. For bits b_0 to b_{31}, the value v is:

$$v = (-1)^{b_{31}} * 2^{(b_{30}b_{29}...b_{23})_2 - 127} * (1.b_{22}b_{21}...b_0)_2 \quad (1)$$

In this format, the most significant bit represents the sign of the value, bits 30 through 23 define the exponent, and bits 22 to 0 make up the mantissa.

As mentioned, the artifacts we are trying to minimize are caused by offsets caused by altering the lower bits of the mantissa. We can obtain an upper bound by finding the maximum offset of value we can cause, which would be by going from all 0s to all 1s or vice versa.

Assuming we are allocating k bits from the mantissa for encoding, the largest difference on a given axis is:

$$\sum_{i=1}^{k} 2^{i-24} * 2^{(b_{30}b_{29}...b_{23})_2 - 127} \quad (2)$$

This error greatly depends on what the exponent is, but meshes are typically modelled around the origin in software such as Blender (Blender Development Team, 2022). For a floating-point vertex coordinate value (denoted as n), the exponent is equal to $2^{\lfloor log_2(n) \rfloor}$, or floored to the nearest power of two. For a float value of n, the maximum error would be:

$$\sum_{i=1}^{k} 2^{i-24} * 2^{\lfloor log_2(n) \rfloor} \quad (3)$$

This makes sense since the maximum error is defined by flipping all bits from 0 to 1 or vice versa. This essentially causes the mantissa to go from 1 to a value very close to 2 (or the other way around), which is then multiplied by the exponent, which is the value floored to the nearest power of two.

This can be done by adding in some arbitrary data with an increasing number of bits until artifacts can be seen. The arbitrary data we chose was the first 19 digits of pi on a loop for all vertices as placeholder for real data. We used 19 digits specifically because that was the most that could fit in a 64 bit integer.

Figure 2 and 3 shows this while encoding a different number of bits on the commonly used armadillo testing mesh. We also note that the armadillo mesh has a bounding box of 127×151×155 units in its local modeling space (as defined by the software, e.g., Blender), centered around the origin.. From a distance in Figure 2, the artifacts become visually distracting at 16 bits per axis and at 20 bits per axis the mesh loses all local detail.

Closer up in Figure 3, they are visible at 12 bits per axis, and add very noticeable bumpiness at 15 bits per axis. From this test, we found empirically that for a mesh centered around the origin, one can readily make use of 10-12 bits per axis. A typical 3D coordinate requires 96 bits—comprising three 32-bit floating-point values, one for each axis. By embedding 10 to 12 bits of metadata per axis, we effectively store each vertex using only 60 to 66 bits for positional data.

4 Encoding Ambient Occlusion

A useful application for these extra bits that can be concisely described is encoding static ambient occlusion (AO). As the name suggests, AO is a shading technique to show the amount that a particular point in space is occluded from the ambient lighting. This essentially darkens areas inside of nooks and crannies.

For this, we encode the AO per vertex and use vertex shading to apply it. For our experiment we first calculate the AO per vertex using MeshLab (Cignoni et al., 2008). The AO calculation method is not critical since our focus is on encoding it, however we note that Meshlab uses a ray tracing method. These AO values range from zero to one, with zero meaning the vertex is fully occluded and should be coloured black, and one meaning it is not occluded at all.

In this case, we know the data will be scalar values between zero and one, so an appropriate way to store this value would be to use fixed point encoding. The number of bits we allocate in each axis determines the fidelity of the AO. For example, using two bits means we have four numerically evenly spaced grayscale values at our disposal. Vertex shading is then used to apply the AO.

Figure 4 shows this in practice with a varying number of allocated bits. The ground truth is the same AO based vertex shading but with a 32 bit floating point number, relating to if the vertices had a colour attribute allocated. With just 6 bits per vertex, the output is virtually imperceptible from the ground truth.

We note that quality may vary depending on the triangulation and resolution of the mesh. For instance, figure 5 shows this applied to a low resolution skull with lighting applied, take note of how the eye sockets

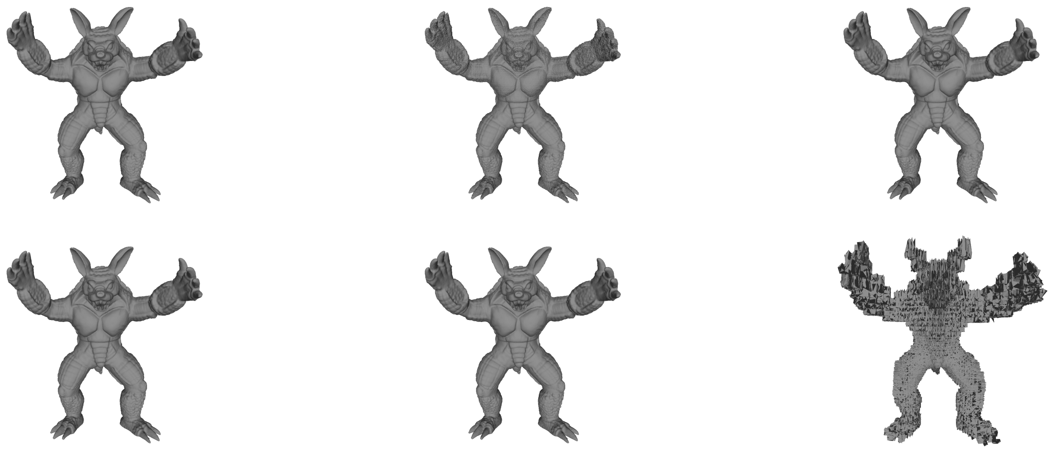


Figure 2: Encoding pi in the bottom 0, 4, 8, 12, 16, 20 bits of vertex positions, left to right.

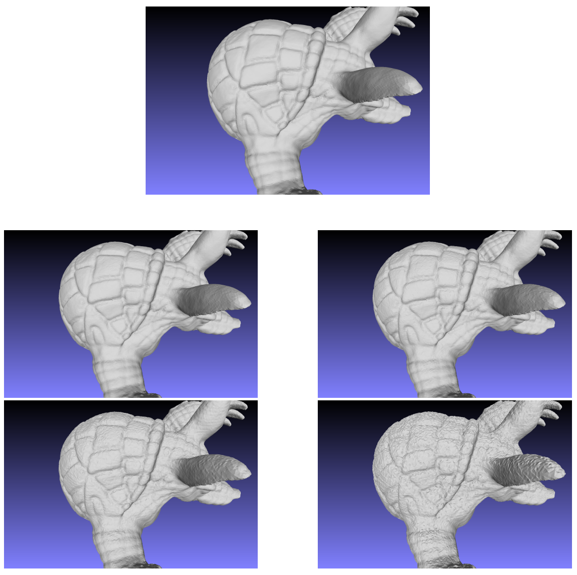


Figure 3: Original mesh (top), and encoding pi in the 12 (top left), 13 (top right), 14 (bottom left), and 15 (bottom right) bits of vertex positions.

and nose darken with AO.

With only a small number of bits per vertex, static AO can be stored using a fixed point value to a level that is imperceptible from its vertex shaded equivalent. Furthermore this encoding does not require many bits per vertex. These two aspects mean that one could use the mesh while completely ignoring the encoded AO, but could always opt in to using it.

While AO is commonly stored in texture channels in modern workflows to decouple shading detail from mesh granularity, our approach demonstrates that vertex-based AO can be embedded without additional attributes or files. This case study is intended as a proof-of-concept for the encoding method rather than a replacement for established practices. The same principle could be applied to other per-vertex data where texture-based storage is not feasible or desirable.

5 Encoding Brittle Fracture

We can also encode information about the internal structure of the mesh and not just its surface funtionality. Specifically, we investigate methods of encoding a prefractured version of the mesh inside the encodable lower bits. This is achieved by storing a set of cutting primitives and procedurally generated noise used to fragment the mesh into different parts. This is done in a way that can imply heterogeneity and anisotropy of the material that the mesh is made of by encoding different areas of the mesh with differing patterns of fracture.

Our method focuses on the prescoring process: allowing an artist to fracture a mesh ahead of time, then instantaneously swapping the prefractured mesh with the original mesh at the time of an impact. Prior work typically involves storing two meshes that must be kept together and synced throughout any development process, but by storing the fracture information in the lower bits of the mesh, we eliminate this issue since it can be extracted at any time when there is a cause for it to break. This eliminates the storage of requirement of the second mesh and removes the logistical upkeep of syncing the two meshes across the development cycle of the application.

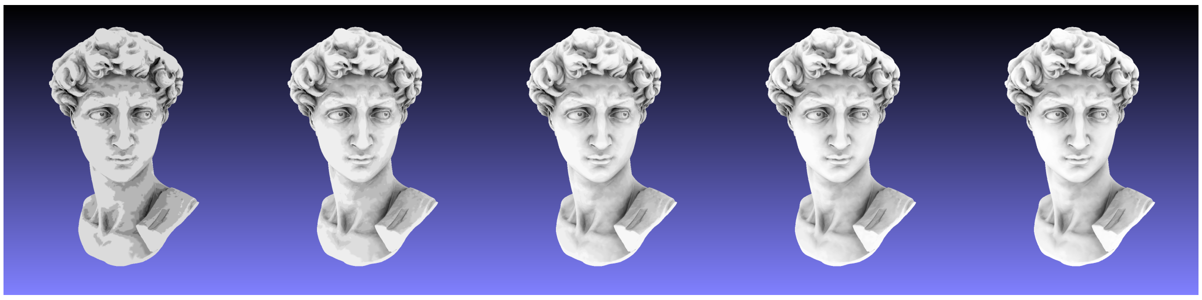


Figure 4: Ambient occlusion encoded in 3, 4, 5, 6 bits per vertex (left to right), and ground truth (right). Note the colour banding in the images on the left.

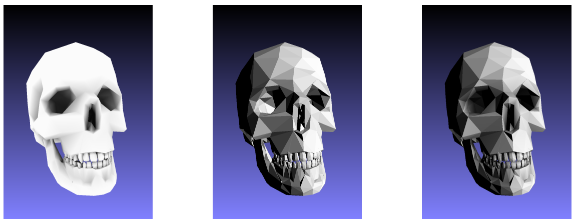


Figure 5: Ambient occlusion encoded with six bits per vertex on a skull. From left to right, AO with no shading, shaded mesh without AO, shaded mesh with AO.

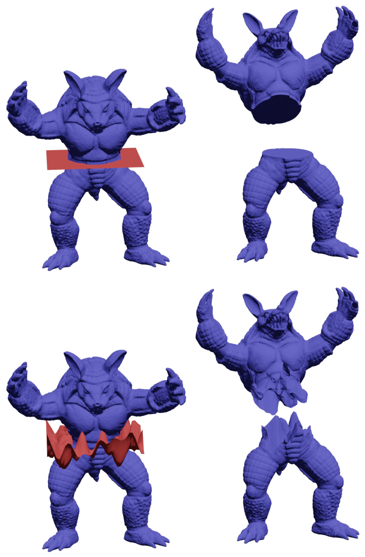


Figure 6: Armadillo cut with a flat plane (top) and with a plane with a displacement map (bottom). Displacement maps can imply an internal structure, and to make the fracture more interesting.

5.1 Methods

As a general overview, we make use of primitive objects, namely planes and cylinders as "cutting meshes" to separate the source mesh into fragments through boolean operations.The internal geometry exposed by fractures is generated during these boolean operations using the MCUT library, which ensures manifold fragments and creates the necessary internal surfaces. This is analogous to the knife tool in Blender (Blender Development Team, 2022), in which the cursor is used to draw a path along an object where afterwards new edges are placed on the geometry, allowing it to be separated.

Simple primitives could be used for this, but would produce cracks and fracture lines along the mesh that are flat and uninteresting. We apply a displacement map on top of these cutting meshes that can make the fracture paths more interesting and help to imply an interior structure. An example of this can be seen in Figure 6. To keep storage requirements low for the encoding, these displacement maps are generated from custom noise functions. This is all encoded in the lower bits of the mesh and can be extracted at any time.

These primitives and noise functions can be stored in very few bits which is in line with our aims. The full format to store this information can be seen in Listing 1, and the following sections will explain the reasoning behind each specific part of the format.

5.2 Primitives

We use planes and cylinders as our cutting meshes since they are simple to encode, and can describe a wide variety of different fracture patterns. For planes, it is as simple as saving the four corner positions. This allows any quadrilateral, with a possible corner by triangulating the four positions in a fixed order, based on the order they are encoded.

Cylinders can be reconstructed by saving the center locations of each circular face, along with the radius of each of them. We also include a quality value, that corresponds to how many vertices to construct the circular face of the cylinders with and add a quality value for the planes where it determines how finely it is subdivided along each axis. Examples of how this affects the geometry can be seen in Figure 7.

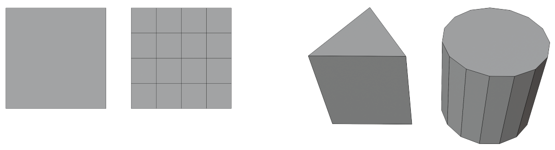


Figure 7: Example of how the quality values effect the geometry of the primitives. The planes have a quality value of 0 (left) and 3 (right). Cylinders have values of 3 (left) and 15 (right).

Figure 8 show how each of these primitives can be used to ”cut out” or fracture a sphere into different fragments. Cuts are performed using MCUT (Chitalu and CutDigital, 2020), and will be described in more detail in Section 5.5.

With this we can evaluate how much memory we need per primitive, in terms of geometry. Planes need four positions or twelve floats, and cylinders require two positions and two radii for eight floats. An immediate way to lower the storage requirement is to notice that the precision of these are not extremely critical, so the 16-bit encoding of a float is used for any fracture information.

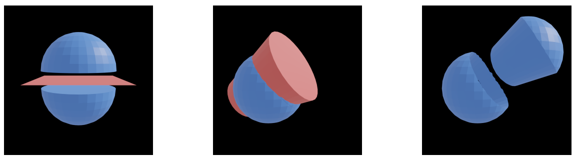


Figure 8: Cuts made on a UV sphere with each primitive. Red indicates the cutting mesh, blue is the source mesh. Requires 241 bits for the plane cut, and 177 bits for the cylinder cut.

5.3 Displacement maps

Creating more interesting fractures beyond using only planes and cylinders would require an immense number of them, most likely exceeding the number of available bits for encoding in the source mesh. A way to add more detail without adding more primitives is to use a displacement map. This is essentially an image that is applied to define how a vertex is offset from its original position, determined by the colour or values in that image. The most common type of displacement map is a heightmap, where a flat plane is raised or lowered based on the value in a grayscale image.

We use displacement maps to modify the cutting meshes themselves. An example of this can be seen in Figure 9 where a grayscale image (left) is used as a displacement map (right) on a plane. Bright areas indicate moving forward along the normal, and darker areas in the opposite direction. A similar concept

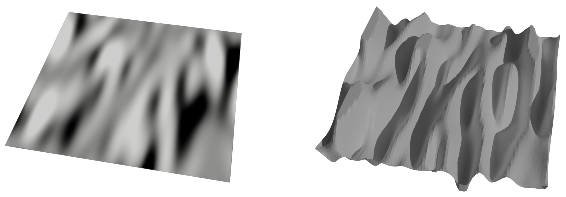


Figure 9: Heightmap used on a plane. Left shows the heightmap image and right is the plane after it is applied.

is applied for cylinders except the image is wrapped around like a soup can label. Bright areas would push outwards and dark areas would shrink inwards.

Procedurally generated noise is used to generate these images since they are infinitely scalable and provide interesting patterns with relatively little artistic skill required. FastNoise2 (Peck, 2020) is used as the noise generation library for two reasons. The first is that it uses a node-based interface and therefore requires no programming knowledge to use. The second is that it has a compact way of storing a node tree in base64. Figure 10 shows an example of the FastNoise2 interface. The noise is made by compounding these mathematical functions together.

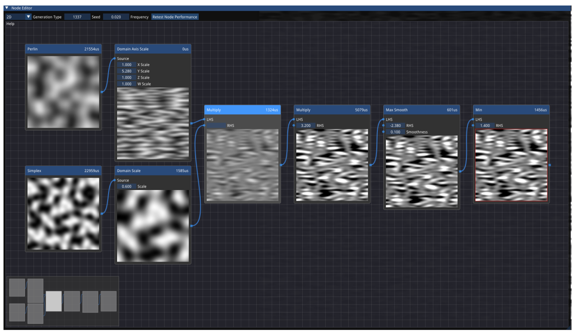
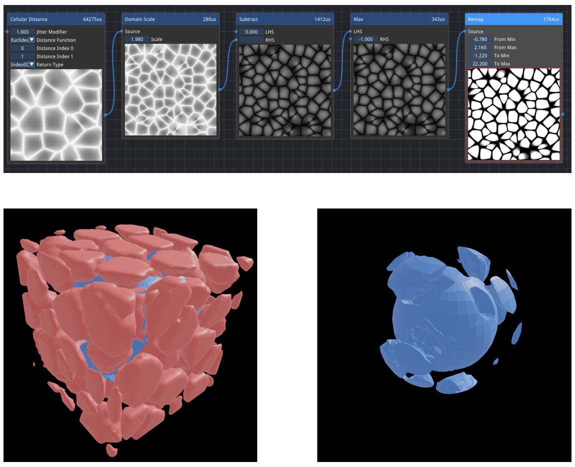


Figure 10: Example node tree in FastNoise2. This example is used as a displacement map in Figure 13 and has a base64 encoding of 56.25 bytes.

Many mathematical functions are centered around zero, for this reason the noise function is then sampled in a 2x2 grid centered at the origin with a domain of $[-1,-1] \rightarrow [1,1]$. A 128x128 image is created using this sampling. This texture is then applied to the mesh, using UV coordinates to sample the texture to offset the mesh along its normal vectors. Since the displacement texture is part of an intermediate step, the resolution could be increased from 128x128 with little extra computational cost, and it does not affect the fracture encoding.

The displacement map only gets applied to the cutting primitive’s vertices, which is the purpose of the quality values mentioned previously. For example, a plane with only four vertices can only be moved at its corners, remaining flat. By subdividing the plane we provide more vertices for the displacment map offset. This can be increased to create more detail in the cut surface at the cost of a more geometrically complex fragment.

FastNoise2's base64 encoding makes analysis of the storage requirement in the mesh simple. For each noise function, we encode the length of the base64 with a 10 bit unsigned integer for a maximum length of 1024, followed by the base64 encoded string with 6 bits per character.


Cellular Distance 64275us
1.000 Jitter Modifier
Euclidea Distance Function
0 Distance Index 0
1 Distance Index 1
Index0D Return Type

Domain Scale 280us
Source
1.980 Scale

Subtract 1412us
0.000 LHS
RHS

Max 343us
LHS
-1.000 RHS

Remap 1764us
Source
-0.780 From Min
2.160 From Max
-1.220 To Min
22.200 To Max

Figure 11: Example of 3D noise used to generate a cutting mesh in an affect somewhat like a fracturing rock (bottom), and noise tree used for the effect (top). Requires 849 bits.

5.4 Volumetric cuts

While it is entirely possible to create any type of fracture with primitives and noise, there are times when it can become cumbersome and unintuitive. For this reason we also include a method of generating a cutting mesh through a 3D volume generated by 3D noise, where a scalar field of points is made and the value at each point indicates a density at that point in space. This density grid can then be passed through a point cloud to mesh algorithm such as marching cubes (Lorensen and Cline, 1987) or poisson surface reconstruction (Kazhdan et al., 2020) or even just the convex hull. We use marching cubes in our implementation. The mesh is then used to cut out a portion of the source object.

We again generate the scalar field of points and it is encoded the same way as the displacement maps. Similar to the displacement maps, we sample centered at the origin in a 2x2x2 cube, or a domain of $[-1,-1,-1] \rightarrow [1,1,1]$. However, slightly more information is needed for how to transform the cutting mesh into the correct placement. We also include three coordinates for the position, one float for the overall scale, and three floats for the Euler rotation. This allows it to be transformed anywhere along the source object and is beneficial if one only wanted to cut specific points of the mesh.

A further value of quality is required, denoting how finely to sample the noise function in the 2x2x2 grid. This was not needed on the displacement map because even if the resolution was increased beyond 128x128, it would not affect the output geometry of the cutting mesh, and therefore the resulting fragments. This was instead controlled through the quality values of the cutting primitives themselves. However, in this case the quality of the scalar field will directly affect the resolution of the cutting mesh, and therefore the resulting fragments. We would like to retain some control of the geometrical complexity of the output, so we include a quality value in the encoding denoting how many points to place within an axis of the 2x2x2 grid.

An example of this can be seen in Figure 11, where 3D noise is used to cut out portions of a sphere in a way that would be very difficult using primitives.

5.5 Boolean operations

At this point all the components needed for the fracture are defined. All that is left is to use our cutting meshes to actually separate the source object into fragments. To perform the cutting operations we make use of the MCUT library (Chitalu and CutDigital, 2020), which is a robust slicing and partitioning library of any manifold object constructed out of simple polygons. A good feature of this library to note is that on an operation between two meshes, the connectivity and complexity of the output meshes remain unchanged, except for where the intersection took place. This allows for better control over the fractured mesh, unlike some physics-based methods where mesh topology is recalculated at each timestep to account for internal stresses and fractures.

In the cutting process, the source mesh is separated using each cutting mesh, in the order they are encoded. MCUT only allows a mesh to be cut by a single other mesh at a time, so whenever the source mesh is separated, those fragments are iteratively reintroduced into the cutting process to be further separated by the remaining cutting meshes, removing the original mesh those fragments were made from.

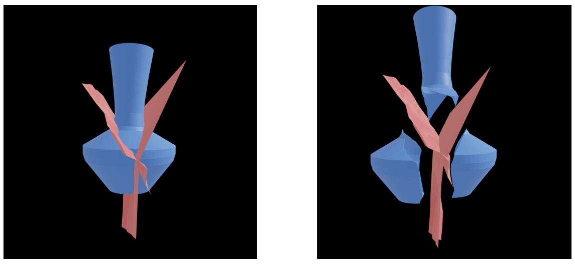


Figure 12: Vase cracked into three pieces. Requires 1594 bits.

5.6 Final Encoding Structure

The final encoding structure is as follows. The noises are encoded first, beginning with the number of noises as a four bit integer, for a total of 15 bits. These are one indexed, so that a primitive can indicate to apply no displacement map by using zero. After this those noises are encoded as described in Section 5.3. Following this the number of cutting primitives are encoded as an 8 bit integer, followed by the plane and cylinder encodings described in Section 5.2 in cutting order. Prior to each primitive, one bit is used as an identifier to declare the next shape as a plane (1) or cylinder (0). The volumetric cuts are encoded next. This again begins with the number of volumetric cuts as a 4 bit unsigned integer, followed by the noise encoding as describe in Section 5.4.

Recall from Section 3 that approximately 10–12 bits per axis can be safely repurposed without introducing visible artifacts. Using this as a basis, the total storage requirement for fracture encoding can be expressed in B bits as:

$$B = 225P + 161C + \sum_{n \in N}(6|n| + 10) + \sum_{v \in V}(6|v| + 329) + 20 \tag{4}$$

and P, C are the number of planes and cylinders respectively. N, V represent the displacement noise and volumetric noise encodings. $|n|$ denotes the length of a noise encoding n.

Listing 1: Fracture Encoding format

```
Noise count : 4 bit unsigned integer
[
{
 Length of FastNoise2 Noise encoding:
          10 bit unsigned integer
 Base64 Encoded Noise:
          6 bits per character
}
...
]

Primitive shape count: 12 bit unsigned int
 [
 {
  (Plane encoding)
  Primitive type (0=plane, 1=cylinder):
            1 bit
  Positions of  4 corners in 3D space:
            3*16 bits per corner
  Index of noise for this primitive:
            4 bits
  Noise strength:
            16 bits
  Quality - How much to subdivide plane:
            12 bits
 }
 ...
 {
  (Cylinder encoding)
  Primitive type (0=plane, 1=cylinder):
            1 bit
  Positions of top & bottom circular faces:
            3 * 16 bits per side
  Radii of the top and bottom:
            16 bits per side
  Index of noise for this primitive:
            4 bits
  Noise strength :
            16 bits
  Quality - # vertices for circular faces:
            12 bits
 }
 ...
 ]

Volumetric Cut Count : 4 bit unsigned integer
 [
 {
  Length of FastNoise2 Noise encoding:
            12 bit unsigned integer
  Base64 Encoded Noise:
            6 bits per character
  Quality: 16 bit unsigned integer
  Position of volume: 16 * 3 bits
  Scale of volume: 16 bits
  Rotation of volume: 16 * 3 bits
 }
 ...
 ]
```

5.7 Decoding Process

The encoding relies on a predefined, deterministic structure embedded in the lower bits of vertex positions. Each segment of bits follows a strict order: starting with the number of noise functions, then identifiers for cutting primitives (planes or cylinders) and their parameters. A single-bit flag distinguishes planes from cylinders, and volumetric cuts and displacement maps are indexed similarly. Because the format is fixed, a decoder can sequentially read these bits and reconstruct all fracture information without ambiguity, even though no separate metadata fields exist.

5.8 Results

With cutting meshes made from primitives, it is possible to indirectly store a fractured version of a source mesh compactly in the lower bits, as shown in Figures 6, 8, 11 and 12. Further detail and implicit structure of the material of the object can be shown through displacement maps applied to the cutting meshes. Figure 13 shows this with a single plane and a single noise function, creating a similar result to a tree trunk or log snapping and falling over.

It is also possible to mimic physically based methods, assuming the initial conditions and stresses are known. Figure 14 show a cube broken under the three modes of fracture, given an initial crack.

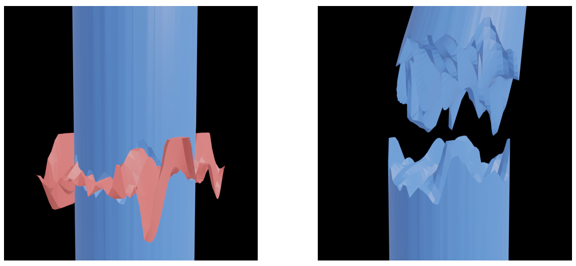


Figure 13: Fracture that mimics a wooden log or tree trunk cracking, exposing the wood grain. Requires 709 bits.

Also by simply not cutting through an area, it is also possible to imply that some parts of the object are fragile while others are not. In Figure 15 a light bulb is broken with increasingly more fragments. However, the cap that screws into the socket is never cut through, implicitly showing that it is made of a stronger material than the bulb itself.

5.9 Discussion

This method of storing cutting meshes and displacement maps in the lower bits can be used to create interesting and intricate fracture patterns. They can mimic physically based methods, such as the modes of fractures on cubes, and imply heterogeneity and anisotropy, while maintaining a level of control for the geometrical complexity of the output fragments. Additionally, it removes the logistical upkeep of synchronizing the mesh and its prefractured counterparts across development.

An alternative approach would be to store additional attributes or apply light compression to the mesh. Compression could indeed hide its processing cost behind I/O operations, and extra attributes offer flexibility for arbitrary metadata. However, these methods introduce trade-offs that our approach aims to avoid. Compression requires custom decoding and may complicate interoperability with standard tools, while extra attributes increase file size and often require format-specific extensions. In contrast, embedding data in the lower bits preserves compatibility with existing formats and workflows without additional files or parsers. That said, the “bit reservoir” is finite, and for highly detailed meshes or large metadata requirements, compression or auxiliary attributes may become more practical. Future work could explore hybrid solutions that combine embedded bits with lightweight external metadata for maximum flexibility.

We also note that for scenes with very large spatial extents and extremely fine details, the available “bit reservoir” may be insufficient without introducing visible artifacts. For example, encoding coordinates for a 1000 m × 1000 m × 100 m environment with millimeter-level granularity consumes nearly all mantissa bits, leaving little room for metadata. In such cases, modern formats that support additional attributes or lightweight compression may offer more flexibility without compromising precision. Our approach is primarily intended for scenarios where mesh scale and detail allow for safe repurposing of lower bits, and where compatibility with existing file structures is a priority. Future work could explore hybrid solutions that combine embedded bits with external attributes for cases requiring extreme precision.

It is important to note that the number of bits that can be safely repurposed is not universal. The acceptable precision loss depends on the spatial extent of the mesh and the smallest geometric features it contains. Larger meshes or those with fine details may require more positional accuracy, reducing the number of bits available for encoding. Conversely, smaller or less detailed meshes allow for more flexibility. This variability means that any implementation should consider mesh scale and detail before deciding how many bits to allocate for metadata.

Many of the choices in the encoding format were made to keep it simple and flexible, but can be improved upon. For example, the noise count uses four bits for a maximum of 15 noises. This could have been raised or lowered depending on if the amount of noise patterns are deemed more or less important.

One consideration is the impact of internal geometry on rendering performance. Unlike conventional prescoring approaches that swap meshes, our

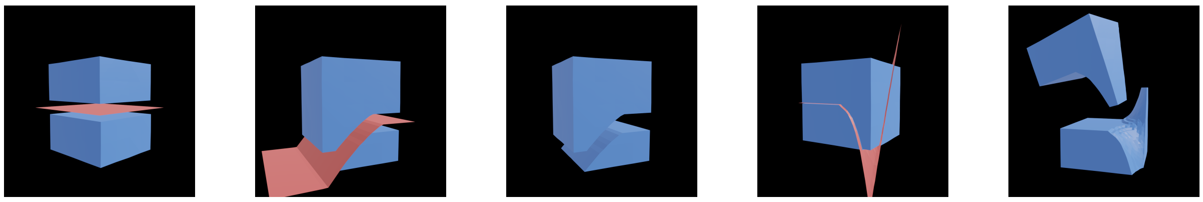


Figure 14: First three modes of fracture I, II, III from left to right, mimicked by a single cutting plane and a displacement map. Requires 241, 877, and 997 bits respectively. Cube was subdivided to 40 vertices to have enough encodable bits.

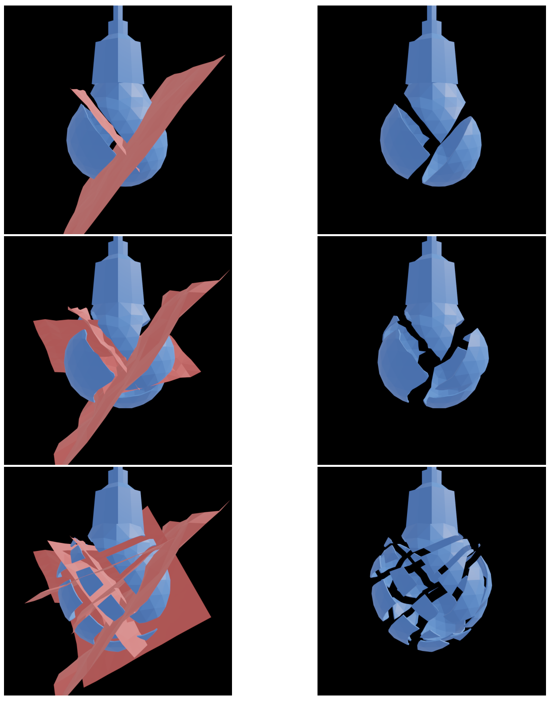


Figure 15: Light bulb with increasingly more fragments. Requires 934, 1384, and 2284 bits respectively.

method retains the fractured geometry within the original mesh. This can increase vertex and polygon counts, adding load to the rendering pipeline. However, this overhead is only incurred when the fracture is activated, and can be mitigated by techniques such as level-of-detail (LOD) management or deferred mesh subdivision. Future work could explore dynamic loading of fracture data to minimize runtime cost.

We also note that embedded fracturing could be taken a whole other direction where instead of prescoring, the fracture could instead be physically based. We could use 3D noise to encode the internal structure of the mesh such as its strength and toughness through space. With all the initial conditions known, it could be passed though a physically based fracturing simulation based on a given impact. This would provide a more realistic output that also responds to user input, perhaps at the cost of more expensive computation.

6 Limitations

There are a few limitations of our encoding method. We are limited by the number of bits we can encode per vertex so there is a maximum amount of information we can encode in the entire mesh. We are free to use this storage in any format we choose but we are limited by the total storage unless more vertices are added.

We are also restricted to file formats that do not truncate or compress the mesh since we rely on the lower bits of the positional data. This means we're mostly restricted to binary formatted files.

Another limitation is the fragility of the embedded metadata in typical workflows. Most 3D tools and pipelines are unaware of the encoded information, so operations such as re-exporting, vertex welding, or precision reduction can alter the least significant bits. This may lead to inconsistencies or unexpected behavior when software that interprets the embedded data attempts to decode it. Therefore, this method is best suited for controlled environments where mesh integrity can be guaranteed, or where encoding and decoding occur within the same toolchain.

7 Conclusions And Future Work

In summary, we have introduced and investigated the idea of encoding additional functionality into the lower bits of the positional data of 3D vertices. We have found that with approximately 10 bits per axis per vertex, arbitrary data can be encoded with no perceptible visual artifacts.

In contrast to creating a new file format, this method of encoding metadata in the lower bits has better accessibility. No custom parser is required to open and manipulate the mesh and only those who wish to use the metadata need to concern themselves with extracting it. Furthermore, in contrast to storing the metadata in a separate file, there are benefits to the encoding method. For example, it was mentioned in the brittle fracture section that it lowers the logistical upkeep of syncing two different files. But in general, having one file with all the information on the mesh is more convenient than two files that cannot be separated. With the encoding method this also comes at no extra storage cost.

For our first application, we have shown that vertex based ambient occlusion can be encoded with as few as six bits per vertex for a result that is visually imperceptible from using vertex colours. We have also shown that it is possible to store a prefractured version of the mesh inside the object itself.

But there are many other applications of our central approach of adding functionality directly into the mesh. For example, there are ways to add accessibility to the mesh, such as detecting vertices vital to the silhouette and encoding a value to increase the contrast on them for the visually impaired. Other possibilities include encoding mesh similarity and information for machine learning based mesh enhancements. Another promising direction is embedding sound information directly in the mesh, in cases where the object is struck or dropped, for instance.

ACKNOWLEDGEMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Discovery Grant. The authors gratefully acknowledge this funding, which made this work possible.

REFERENCES

Benedens, O. and Busch, C. (2000). Towards Blind Detection of Robust Watermarks in Polygonal Models. *Computer Graphics Forum*, 19(3):199–208.

Blender Development Team (2022). Blender (version 3.1.0). `https://www.blender.org`.

Cayre, F. and Macq, B. (2003). Data hiding on 3-d triangle meshes. *Signal Processing, IEEE Transactions on*, 51:939 – 949.

Chitalu, F. and CutDigital (2020). Mcut (version 1.3.0). `https://cutdigital.github.io/mcut.site`.

Chitalu, F. M., Miao, Q., Subr, K., and Komura, T. (2020). Displacement-correlated xfem for simulating brittle fracture. *Computer Graphics Forum*, 39(2):569–583.

Chou, P. H. and Meng, T. H. (2002). Vertex Data Compression through Vector Quantization. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):373–382.

Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., and Ranzuglia, G. (2008). MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference*.

Deering, M. (1995). Geometry compression. In *ACM SIGGRAPH*, pages 13–20.

Kaufmann, P., Martin, S., Botsch, M., Grinspun, E., and Gross, M. (2009). Enrichment textures for detailed cutting of shells. *ACM Trans. Graph.*, 28(3).

Kazhdan, M., Chuang, M., Rusinkiewicz, S., and Hoppe, H. (2020). Poisson surface reconstruction with envelope constraints. *Computer Graphics Forum*, 39(5).

Langer, M. S. and Bülthoff, H. H. (2000). Depth discrimination from shading under diffuse lighting. *Perception*, 29(6):649–660. PMID: 11040949.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *ACM SIGGRAPH*, page 163–169, New York.

McGuire, M., Osman, B., Bukowski, M., and Hennessy, P. (2011). The alchemy screen-space ambient obscurance algorithm. In *ACM SIGGRAPH Symposium on High Performance Graphics*, pages 25–32, Vancouver.

Miller, G. (1994). Efficient algorithms for local and global accessibility shading. In *Proceedings of ACM SIGGRAPH*, page 319–326, New York.

Mittring, M. (2007). Finding next gen - cryengine2 (course notes). New York, NY, USA 97-121.

Müller, M., Chentanez, N., and Kim, T.-Y. (2013). Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, 32(4).

O'Brien, J. F., Bargteil, A. W., and Hodgins, J. K. (2002). Graphical modeling and animation of ductile fracture. *ACM Trans. Graph.*, 21(3):291–294.

Peck, J. (2020). Fastnoise2. `https://github.com/Auburn/FastNoise2`.

Schvartzman, S. C. and Otaduy, M. A. (2014). Physics-aware voronoi fracture with example-based acceleration. *Journal of Computer Graphics Techniques (JCGT)*, 3(3):35–54.

Sifakis, E., Der, K. G., and Fedkiw, R. (2007). Arbitrary cutting of deformable tetrahedralized objects. In *Symposium on Computer Animation*, page 73–80.

Wang, K., Lavoue, G., Denis, F., and Baskurt, A. (2008). A Comprehensive Survey on Three-Dimensional Mesh Watermarking. *IEEE Transactions on Multimedia*, 10(8):1513–1527.