Mesh Representation 1

http://www.mushware.com/
File generated 2005-04-26 11:15
DRAFT: This document has not been reviewed and may contain errors.

Introduction

This document describes the representation of a four dimensional object using a mesh of connected vertices.

Problem

The 4D form of objects is to be represented through linked groups of vectors, known here as the mesh. The organisation should support the required operations to be performed, which are:

- Subdivision. Subdivision generates a new mesh at higher resolution than the origianl. Must support variable level of detail.
- Rendering. Purpose is to generate the triangle list for the renderer. Minimal alteration is useful – a method to calculate new positions without regenerating the list. Calculation should be pushed into the graphics driver where possible.
- Animation. Hierarchical objects connected by rotating joints, or some other method.
- Collision checking. Division into reasonable-sized chunks for collision checking, each with a bounding region.
- Order-dependent transparency. Rough facet order should be obtainable.
- Support chunk-wise occlusion test for rendering speed.
- Support ‘directional’ information, for determination of whether sides of a face are inside or outside of an object.

Analysis

We should find the simplest representation of the mesh. From this, other representations, such as the facet list for the renderer, can be generated.

Element	**Description**
Vertices	The position of a point on the mesh as a 4-vector
Faces	A hyperface, with extent in three dimensions whilst being flat in the other[1].
Facet	A 2D facet of the 3D face. Analogous to an edge in 3D; drawing only facets is similar to drawing 3D in wireframe, i.e. not realistic.
Edge	A line bounding a facet.
Normal	Relevant to a face only. In 4D three independent vectors are required to generate the normal vector. A pair of vectors have a plane which is orthogonal to them. The normal may be implied by predefined vertex ordering.
Texture coordinates	Dependent on the rendering technique. If facets are rendered as polygons, each facet needs at least two texture coordinates at each point. If faces are rendered directly, at least three texture

[1] There’s no strict requirement that a face be flat, in the same way that a square face in 3D need not have all of its vertices in the same plane. However non-flat faces may complicate the renderer.

	coordinates are required.
Material properties	Material properties stored on a per-face or per-object basis. Smaller scale details must be stored within the textures. Material properties specify which textures should be used and how.
Inside/outside	Records which direction along the normal vector is inside and which outside of the object. This information can be implied by vertex ordering.
Edge smoothness	To what degree an interface between two faces should be smoothed when subdividing.

Table 1: Basic mesh elements

Derived element	Description
Subdivided mesh	A mesh generated by subdividing the original mesh.
Render list	The list of coordinates and state changes to be passed to the renderer.
Collision hull	A simplified mesh used for computing whether objects intersect.
Occlusion render list	A simplified mesh used to test whether an object is visible using OpenGL's occlusion test method.
Chunking	Grouping of faces into a hierarchy of adjacent sets, to aid collision and occlusion tests.
Ordering	Order information for depth sorting.
Bounding regions	Each chunk has a bounding region to be used for culling.

Table 2: Elements derived from the mesh

Subdivision

Subdivision generates new vertices with new connectivity. It presents the following difficulties:

- Subdivision generates an entirely new set of vertices and normals.
- Level of detail (LOD) algorithms require that one part of the mesh may be more finely subdivided than an adjacent part. If large objects are present, i.e. with sections both near to and far from the observer, chunks with differing levels of subdivision my need to be patched together.
- Vertices beyond the current face are required to subdivide smoothly.
- The polygon and vertex count grows geometrically; four fold for each cycle of subdivision.

Collision tests

Collision tests have different requirements to rendering.

- In a multiplayer game only the server need perform collision checking.
- Collision tests do not require access to the renderer.
- The results of some tests may be required before rendering, so that objects are not drawn falling through the floor, etc. This leads to two type of test. Results of projectile-hits-object type tests may not be required before rendering.
- The collision volume may not be the same as the visual volume. It may be simplified, or made larger to make the game easier.

Proposed solution

Firstly the pipeline must be established.

Stage	**Description**
Gather data	A complete set of objects and views is assembled. The following stages take place for each view. Some views may need to be rendered before others, e.g. for environment and shadow mapping.
Pre-collision transform	The renderer can reuse these transforms, and also the sort lists generated for collision checking.
Pre-cull transform	Perform the necessary transforms required for the geometric cull; usually a transform to eye coordinates in 3D.
Geometric cull	Remove objects and chunks from the scene where it is efficient to do so, using geometric tests on bounding boxes and distance.
Occlusion render and cull	Draw large, near objects and perform occlusion tests on distant ones. May not always be necessary. If used to cull texture jobs, no further tests will be required once a particular texture is known to be needed.
Renderer select	The presence of the following actions depends on what type of output detail is required, e.g. environment map, render to texture, final render, etc.
Level of detail	Each chunk is tagged with the level of detail required.
Reuse	Previous jobs where output is close enough are passed for rendering directly.
Chunking	Work chunks are generated for the worker thread(s). Each thread must have a complete set of data to work with, and ideally results should not be calculated mote than once. Render jobs should not be processed in the wrong order, and texture jobs should be despatched first.
Subdivision	The mesh is subdivided as necessary for the level of detail required. It is important to reuse previous work where possible.
Subdivision culling	Some primitives may be discarded during subdivision.
Render job generation	Generate a job definition for the renderer and despatch the job.
Completion	When all chunks are complete, signal the renderer.

Table 3: Rendering pipeline

Stage	**Description**
Pre-collision transform	Perform the necessary transforms required for collision checking; usually a transform to world space in 4D. Sort lists, if the renderer uses them, are generated now, before collision checking proceeds on its own.
Sort	Update sort lists as required.
Candidate sweep and cull	Choose object pairs using the sort list and their extents. Where pairs have known results, use them.
Bounding box cull	If bounding boxes exclude objects, cull the pair.
Mesh cull	Test for intersection using a detailed mesh test.
Report collisions	Return a collision list for further processing.

Table 4: Collision pipeline

To support this pipeline we introduces a new type of mesh data; cached results that persist from frame to frame to make overall calculation more efficient. These are:

Persistent element	Description
Collision test results	Results of collision tests between two objects. If a distance between the two can be calculated, this will be still valid if the objects have not moved further than that.
Sort orders	Object orders maintained for collision tests.
World coordinates	Useful for static objects.
Geometrically culled objects	Objects which are still excluded from the view because, e.g. their angular displacement was too great for them to be visible this time.
Renderer jobs	Entire sets of rendering information for non-animated objects.
Subdivided meshes	Complete meshes which can be completely or partially (e.g. just texture coordinates) reused.

Table 5: Persistent data

Minimal mesh representation

The minimal representation contains just the information needed to define the mesh and no other.

Element	Description
Vertices	A single entry for each vertex.
Material reference	Reference to the global material for the mesh.
Texture coordinates	Any number of entries, as required for the faces.
For each face:	
Face type	At least cubic. Prism and tetrahedron also useful.
Vertex list	The three pairs $0 \rightarrow 1$, $0 \rightarrow 2$, $0 \rightarrow 4$ are used in that order to calculate the face normal. These are index values referring to the vertex entries.
Texture coordinate list	A texture coordinate index value for each vertex in the vertex list for this face.
Material reference	Reference to an overriding material for this face, if any.
Edge smoothness	The smoothness of each edge. In this case an edge is the polygon (a quadrilateral in our case) where two faces meet.

Table 6: Data required for the minimal representation

Additional representation

The additional representation contains hints and precalculation that cannot easily be derived:

Element	Description
Chunking	A chunk hierarchy, dividing the object into groups of faces.
Convex hull	A convex hull for each chunk.

Table 7: Data supplied in the additional representation

Derived representation

The derived representation contains:

Element	Description
Normals	Normal vectors for each face.
Connectivity	A list for each vertex, recording all of its neighbours.
Chunk connectivity	A list for each chunk, recording all of its face-wise neighbours.
Bounding regions	The object and each of its chunks and faces have a bounding region.

Table 8: Data generated in the derived representation

Example: Multi-tentacled beast

Description

A particular creature has eight tentacles, two opposed on each axis meeting at a central blob. One is adapted as an eyestalk and the opposite one is stubby and has a sucker. The tentacles can bend in three axes and twist in another three. To animate, each tentacle is algorithmically drawn towards an attractor point.

The creature has a simple control mesh. To simplify, three tentacles are shown in 3D only.

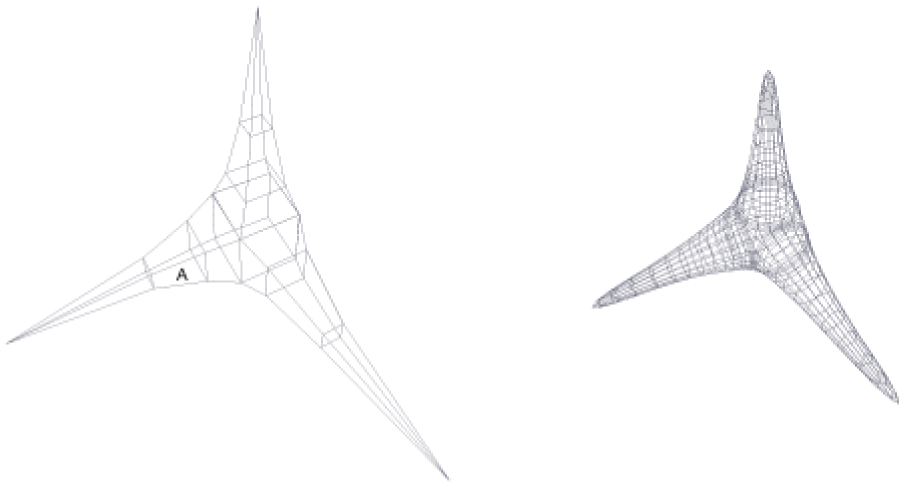

Figure 1: Control and subdivided meshes

In 4D, a cube replaces each square face. The section marked A has four faces, as two are missing where the section joins neighbouring sections. In 4D, this section would be a tesseract with two of its faces missing, leaving six. The shrinkage is an characteristic of the type of subdivision used.

When all tentacles are present the central cube will have no renderable sides, so is not truly a face of the object. If a tentacle is then removed, another face must be introduced to fill the hole.

Generation

This creature can be generated algorithmically. In 3D, the tentacle is defined by its square cross section at four places, which vanishes to a point at the tip. In 4D the cross section will be a cube, similar to a single face of a tesseract. When moving from cross section to cross section, in 3D there are two angular (bending) degrees of freedom, because there are two axes perpendicular to the direction of the tentacle. The tentacle pointing out of the screen can bend up/down and left/right. It can also rotate about its axis. The changes in orientation between neighbouring cross sections must be slight to avoid distortion, but more cross sections can be introduced as required. The tentacle can also change length to some extent, ideally becoming fatter as it shortens.

In 4D, the cross section is a cubic face. There are three angular degrees of freedom, because there are three axes perpendicular to the direction of the tentacle. There are three rotational degrees of freedom. If the tentacle is heading in the *x* direction, rotation can take place in the *yz*, *zw* and *yw* planes.

Initial generation would proceed as follows:

Stage	**Description**
Position	The position of the tentacles must be known. This implies that the position and orientation of each cross section can be calculated.
Initial cross section	The section where the tentacle joins the body is calculated. This could be fixed, or calculated from a mean of adjacent tentacles.
Next cross section	Generate the next cross section along the tentacle, using the position parameters. This could be done by interpolation, or using known bend/rotate values for each segment.
Vertices	Write the new vertices to the definition.
Create the first chunk	In this example each segment of the tentacle's control mesh will be a single chunk.
Generate faces	Add face definitions for the six faces, using the vertices already added. Add texture coordinates as necessary. Mark the edges as smooth.
Repeat	Generate further tentacle chunks as required, for all tentacles. Generate a chunk of each new segment. The end is capped 6 or 7-sided chunk.
Vertex connectivity	Generate the vertex connectivity information for the whole object.
Normal generation	Using the connectivity information, average the normals for each vertex. This is the average value of the normals of each face of which the vertex is a part. Normals of opposite sign can be detected here. Obey edge smoothness values to weight each vertex. For smooth objects, normals can be shared between adjacent faces.
Chunk connectivity	Record the chunk connectivity in the object. This object isn't suitable for a convex hull, but each chunk of the control mesh is suitable for collision tests.
Bounding regions	Calculate centres and bounding radii for each chunk, and for the object as a whole.

Table 9: Stages of initial creature mesh generation

Subsequent updates to the mesh would require:

Stage	**Description**
Position	The position of the tentacles must be known. This implies that the position and orientation of each cross section can be calculated.
Initial cross section	The section where the tentacle joins the body is calculated. This could be fixed, or calculated from a mean of adjacent tentacles.
Next cross section	Generate the next cross section along the tentacle, using the position parameters. This could be done by interpolation, or using known

	bend/rotate values for each segment.
Vertices	Update the vertex definitions.
Normal calculation	Update the normal definitions. This uses the vertex connectivity information previously calculated.
Bounding regions	Recalculate centres and bounding radii for each chunk, and for the object as a whole. This may not be necessary for every movement, but should be an optimised step. Possibly, a recalculation should only be triggered when the bounding volume is required.

Table 10: Stages of subsequent creature mesh generation

Movement

Moving this creature is not trivial. A number of collision checks are required to make sure that a tentacle does not pass through another object, or another part of this object. A simple scheme could be devised:

- When not in contact, the tentacle moved randomly using a consistent-ish bend along its length and minimal rotation.
- When a tentacle touches an object, it recognises it as friendly or nasty.
- For a friendly object, the tentacle moves so as to stay in contact with the objects.
- For nasty objects, the tentacle retracts and reverses its direction of motion.

In this scheme the body of the creature does not move. A walking, slithering, swimming or clambering scheme is also required.

- Zero or one tentacle is the anchor tentacle.
- If zero, the creature swims or walks towards a nearby target, or in large circles.
- If one, the creature waves its tentacles. If a tentacle grabs an object that it hasn't just released, that tentacle becomes the new anchor.
- The anchor tentacle should attempt to coil around what it has just touched.

Version History

1. 2005-04-25: Created with filename mesh-represention1-2005-04-25
2. 2005-04-26: Draft released