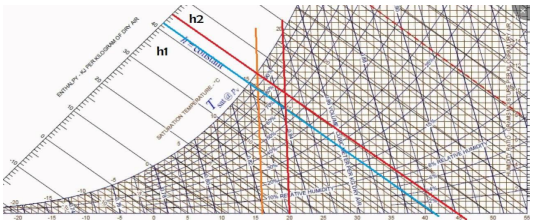
Metaballs & Marching Cubes

Blobby Objects and Isosurfaces

Ben Kenwright



Abstract
Metaballs, also known as blobby objects, are a type of implicit modeling technique. We can think of a metaball as a particle (i.e., a point-mass) surrounded by a density field, where the particle density attribute decreases with distance from the particle position. A surface is implied by taking an isosurface through this density field - the higher the isosurface value, the nearer it will be to the particle. The powerful aspect of metaballs is the way they can be combined. We combine the spherical fields of the metaballs by summing the influences on a given point to create smooth surfaces. Once the field is generated, any scalar field visualization technique can be used to render it (e.g., Marching Cubes). Marching Cubes is an algorithm for rendering isosurfaces in volumetric data. The basic notion is that we can define a voxel(cube) by the pixel values at the eight corners of the cube (in 3D). If one or more pixels of the cube have values less than the user-specified isovalue, and one or more have values are greater than this value, we know the voxel must contribute some component to the isosurface. Then we determine which edges of the cube intersects the isosurface and create triangular patches which divides up the cube into regions to represent the isosurface. Then connecting the patches from all cubes on the isosurface boundary allows us to create a surface representation.

Keywords
metaballs, 3d, 2d, marching cubes, isosurfaces, fluids, soft-bodies, particle, water, tessellation, volumetric data, field

Title: Metaballs & Marching Cubes: Blobby Objects and Isosurfaces

Contents

Introduction

What are metaballs? Metaballs are, in computer graphics, organic-looking n-dimensional objects. The technique for rendering metaballs was invented by Jim Blinn [1] in the early 1980s. Metaballs largely made their introduction in the 1990's through the demoscene: groups of enthusiastic programmers and artists that aimed to create graphical/musical effects that pushed the known limits of older hardware, such as the Commodore 64 and Amiga. The goal of demosceners was to create audio-visual effects in real-time that would impress viewers and confound other demoscene programmers with how the effect was implemented [2, 3]. *The metaballs effect gained popularity because of its squishy organic look and feel.*

What is the marching cube algorithm? Marching cubes is a computer graphics algorithm, published in the 1987 SIGGRAPH proceedings by Lorensen and Cline [3], for extracting a polygonal mesh of an isosurface from a three-dimensional scalar field (sometimes called voxels). This paper is one of the most cited papers in the computer graphics field. The applications of this algorithm has been applied to multiple fields, including video games, medical visualizations, such as CT and MRI scan data images, and special effects or 3-D modelling with what is usually called metaballs or other metasurfaces. An analogous two-dimensional method is called the **marching squares** algorithm [2, 4].

Why do we need marching cube algorithm? In 3-dimensional virtual environments and visualisation software, we typically represent shapes and objects with triangles. Hence, the ability to efficiently convert a set of points into a triangular mesh is important. The marching cubes algorithm is one such technique for rendering isosurfaces. The basic notion is that we can define a voxel(cube) by eight pixel values for the eight corners of the cube. If one or more pixels of a cube have values less than the user-specified isovalue, and one or more have values greater than this value, we know that the voxel contributes to the isosurface. By determining which edges of the cube are intersected the isosurface, we can create a triangular patch which divides up the cube between the regions inside and outside the isosurface. Connecting the patches from all cubes allows to formulate a triangulated representation of the isosurface boundary surface.

Examples Applications of metaballs and marching cubes:

- ✔ fluid simulations (e.g., jugs of water and the sea, such as, smoothed particle hydrodynamics)
- ✔ soft-body systems composed of point-masses
- ✔ x-ray data (i.e., large arrays of scattered points) [5]
- ✔ procedural terrain
- ✔ tessellation algorithms

1. Overview

This article will focus entirely on 2D and 3D metaballs and isosurfaces. Although an isosurface generally refers to a 3D space, we will show that it can very easily adapted to different dimensions. Simply for our purpose, an isosurface, is a surface created by applying one or more functions to obtain the scalar value for a position in space (for example, the scalar strength of any point in a 3-dimensional potential field) The different sections include:

- we discuss the concept and implementation of metaballs and isosurfaces (i.e., cube marching algorithm)
- we examine the current applications of isosurfaces in the video game and graphic industries
- we investigate the performance issues involved with isosurfaces, and some optimization and approximation enhancements

2. Metaball Mathematics

Scalar Potential Field Metaballs are described using the implicit Equation 1 below:

$$\sum_{n=1}^{k} \frac{s_n}{||m_n - p||^g} > r \tag{1}$$

where m_n is location of metaball number n, s_n is size of metaball number n, k is number of balls, g 'goo'-factor, which affects the way how metaballs are drawn, r is the threshold for metaballs, p is the place vector, and $||x||$ indicates the magnitude (length) of vector x.

Normals Metaballs are described by a force field, as shown in Equation 1, so normal for any given point is easy to calculate with the help of gradient, as shown below in Equation 2:

$$\begin{aligned} \text{normal} &= \nabla \sum_{n=1}^{k} \frac{s_n}{||m_n - p||^g} \\ &= \sum_{n=1}^{k} (-g)(s_n) \frac{m_n - p}{||m_n - p||^{2+g}} \end{aligned} \tag{2}$$

3. Marching Cube Algorithm Details

Concept There are two major components of the marching cube algorithm. The first is deciding how to define the section or sections of the surface which chops up an individual cube. If we classify each corner as either being below or above the isovalue, there are '256' possible configurations of corner classifications. Two of these are trivial; where all points are inside or outside the cube does not contribute to the isosurface. For all other configurations we need to determine where, along each cube edge, the isosurface crosses, and use these edge intersection points to create one or more triangular patches for the isosurface.

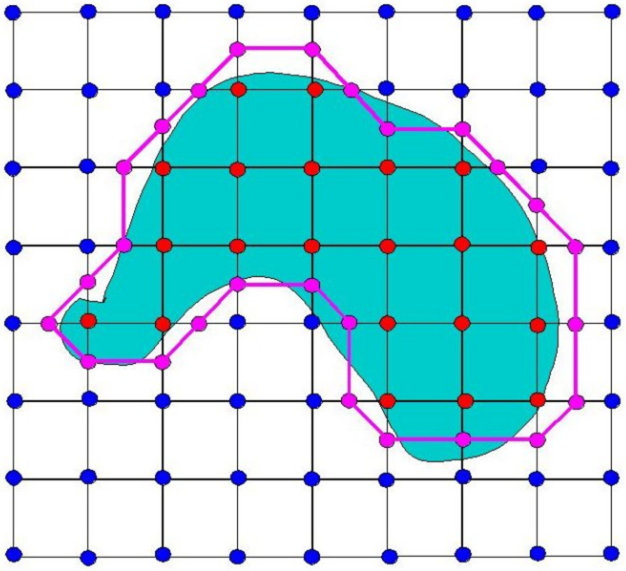


Figure 3. Concept 2D - Help visualize the concept of cube marching in 2D.

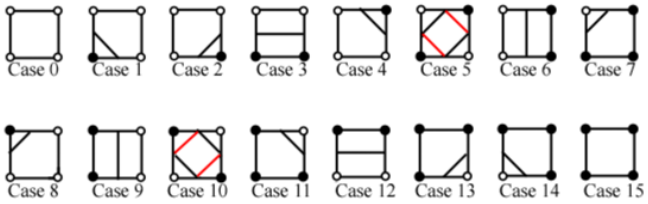


Figure 4. Possibilities for 2D - Applying the technique to a simple 2D example, shown in Figure 3, we can categorize the different surface values for the 16 possible solutions.

Number of Possibilities If you account for symmetries, there are really only 14 unique configurations in the 254 possibilities. When there is only one corner less than the isovalue, this forms a single triangle which intersects the edges which meet at this corner, with the patch normal facing away from the corner. Obviously there are 8 related configurations of this sort. By reversing the normal we get 8 configurations which have 7 corners less than the isovalue. We don't consider these really unique, however. For configurations with 2 corners less than the isovalue, there are 3 unique configurations, depending on whether the corners belong to the same

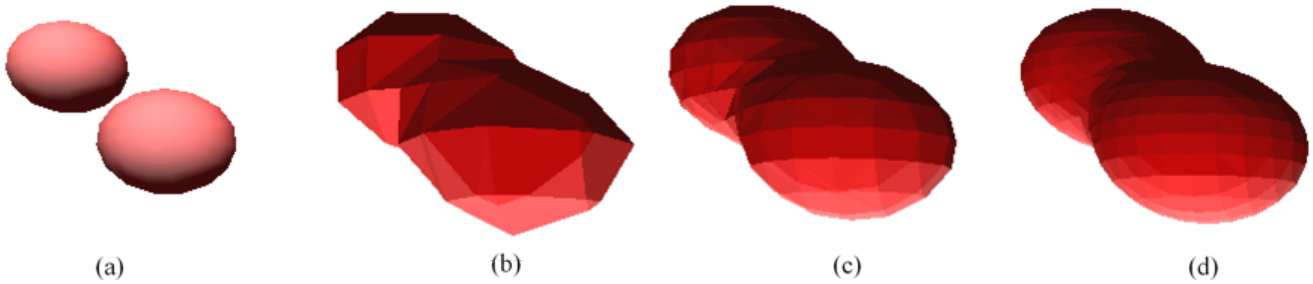


Figure 1. Metaballs and Marching Cubes - Simple test case showing two metaballs (i.e., spheres with specific ratio) and the generated implicit surface. (a) Two spheres, (b) grid resolution of 10x10, (c) grid resolution of 20x20, and (d) grid resolution of 30x30.

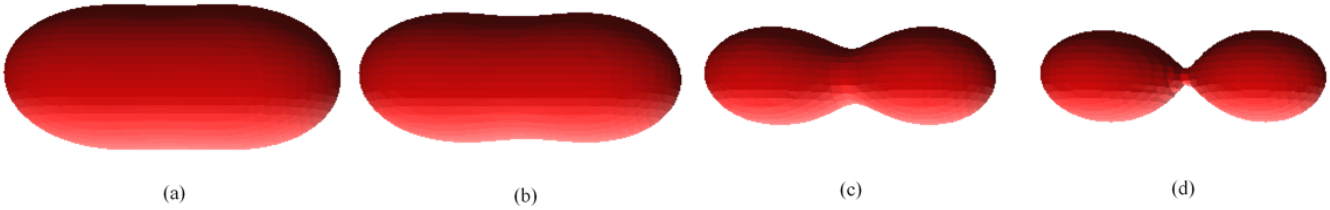


Figure 2. Goo-Factor - Visual effect of varying the goo-factor, in Equation 1, (a) 0.9, (b) 1.0, (c) 1.5, and (d) 1.8.

edge, belong the same face of the cube, or are diagonally positioned relative to each other. For configurations with 3 corners less than the isovalue, there are again 3 unique configurations, depending on whether there are 0, 1, or 2 shared edges (2 shared edges gives you an 'L' shape). There are 7 unique configurations when you have 4 corners less than the isovalue, depending on whether there are 0, 2, 3 (3 variants on this one), or 4 shared edges.

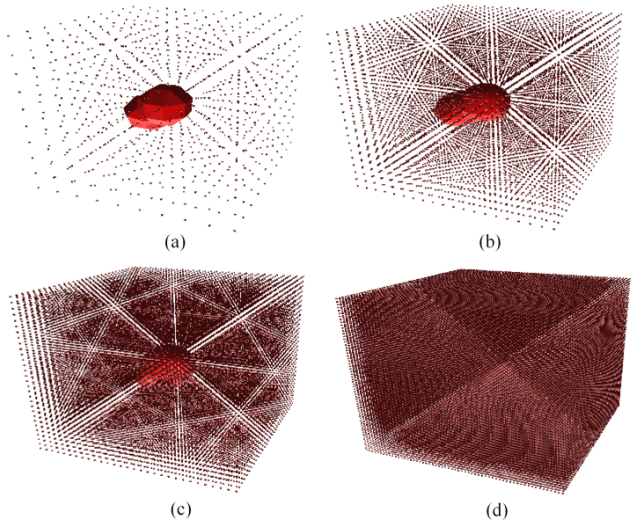


Figure 5. Grid Points - Displaying the grid sample points. (a) 10x10, (b) 20x20, (c) 30x30, and (d) 60x60.

Weights Each of the non-trivial configurations results in between 1 and 4 triangles being added to the isosurface. The actual vertices themselves can be computed by **interpolation** along edges, or default their location to the middle of the edge. The interpolated locations will obviously give better shading calculations and smoother surfaces.

Processing Now that we can create surface patches for a **single voxel**, we can apply this process to the entire volume. We can process the volume in slabs, where each slab is comprised of 2 slices of pixels. We can either treat each cube independently, or we can propogate edge intersections between cubes which share the edges. This sharing can also be done between adjacent slabs, which increases storage and complexity a bit, but saves in computation time. The sharing of edge/vertex information also results in a more compact model, and one that is more amenable to interpolated shading.

Algorithm Steps An uncomplicated step-by-step implementation to create a marching cube isosurface based on a linear grid, is:

1. construct a grid array (e.g., 3 dimensional array of doubles)
2. using Equation 1 calculate the potential field value for each point in the grid array (i.e., influence of each metaball)
3. iterate through all the points constructing a cube region (i.e.,x to x+1, y to y+1 and z to z+1)
4. for each cube region calculate the triangles

4. Further Work

Once you understand the concept of metaballs and the marching cube algorithm, you should be able to expand the concept and create a range of exciting modifications:

- ☑ Large number of metaballs floating around in real-time
- ☐ Optimize for multi-threading and the GPU (e.g., OpenCL and CUDA)
- ☐ Non-linear partitioning of the region
- ☐ Tessellating the surface progressively (i.e., recursively subdividing the voxels)
- ☐ Apply 'Smoothed Particle Hydrodynamics (SPH)' to the metaball movement in combination with the cube marching isosurface calculations to create fluids
- ☐ Move beyond 'squares' towards other shapes, for example, tetrahedron marching - to generate the isosurface

Acknowledgements

We would like to thank all the readers for taking time out of their busy schedules to provide valuable and constructive feedback to make this article more concise, informative, and correct. However, we would be pleased to hear your views on the following:

- Is the article clear to follow?
- Are the examples and tasks achievable?
- Do you understand the objects?
- Did we missed anything?
- Any surprises?

The article provide a basic introduction to getting started with metaballs and the cube marching algorithm. So if you can provide any advice, tips, or hints from your own exploration and development, that you think would be indispensable for a student's learning and understanding, please don't hesitate to contact us so that we can make amendments and incorporate them into future updates.

Recommended Reading

Level Set Methods and Dynamic Implicit Surfaces (Applied Mathematical Sciences), by Stanley Osher, Ronald Fedkiw, Springer, ISBN: 978-0387954820

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

References

[1] James F Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics (TOG)*, 1(3):235–256, 1982. 1

[2] Stanley Osher Ronald Fedkiw. Level set methods and dynamic implicit surfaces. 2003. 1

[3] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM Siggraph Computer Graphics*, volume 21, pages 163–169. ACM, 1987. 1

[4] André Guéziec and Robert Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *Visualization and Computer Graphics, IEEE Transactions on*, 1(4):328–342, 1995. 1

[5] Bradley A Payne and Arthur W Toga. Surface mapping brain function on 3d models. *Computer Graphics and Applications, IEEE*, 10(5):33–41, 1990. 2

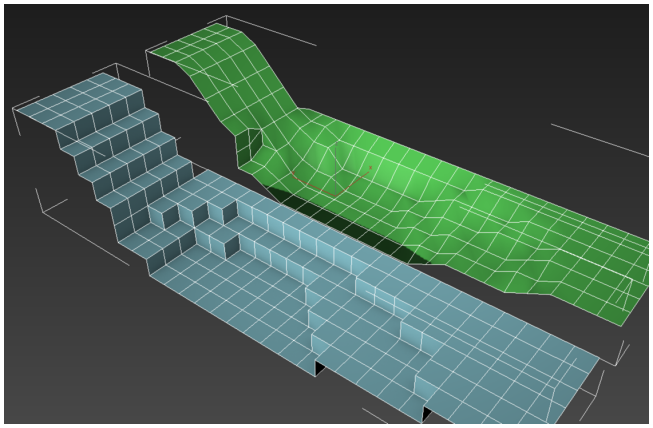


Figure 6. Terrain Cube Marching - Applying the cube marching algorithm to a voxel terrain.

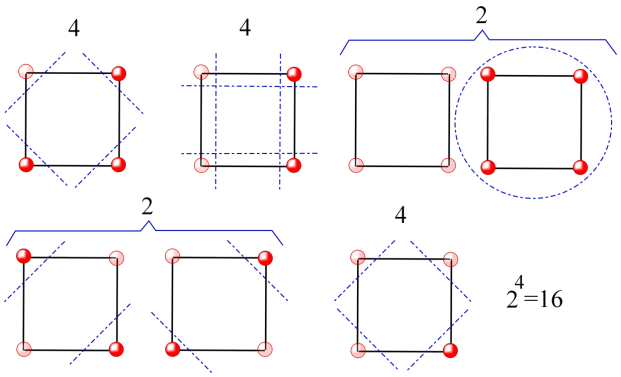


Figure 7. Marching Square (i.e., 2D Marching Cube) - Different combinations.

1. Appendix

Listing 1. Example implementation of the marching cube algorithm.

```
typedef struct {
  XYZ p[3];
} TRIANGLE;

typedef struct {
  XYZ p[8];
  double val[8];
} GRIDCELL;

/*
  Given a grid cell and an isolevel, calculate the triangular
```

```
  facets required to represent the isosurface through the cell.
  Return the number of triangular facets, the array "triangles"
  will be loaded up with the vertices at most 5 triangular facets.
  0 will be returned if the grid cell is either totally above
  of totally below the isolevel.
*/
int Polygonise(GRIDCELL grid,double isolevel,TRIANGLE *triangles)
{
  int i,ntriang;
  int cubeindex;
  XYZ vertlist[12];

int edgeTable[256]={
0x0  , 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c,
0x80c, 0x905, 0xa0f, 0xb06, 0xc0a, 0xd03, 0xe09, 0xf00,
0x190, 0x99 , 0x393, 0x29a, 0x596, 0x49f, 0x795, 0x69c,
0x99c, 0x895, 0xb9f, 0xa96, 0xd9a, 0xc93, 0xf99, 0xe90,
0x230, 0x339, 0x33 , 0x13a, 0x636, 0x73f, 0x435, 0x53c,
0xa3c, 0xb35, 0x83f, 0x936, 0xe3a, 0xf33, 0xc39, 0xd30,
0x3a0, 0x2a9, 0x1a3, 0xaa , 0x7a6, 0x6af, 0x5a5, 0x4ac,
0xbac, 0xaa5, 0x9af, 0x8a6, 0xfaa, 0xea3, 0xda9, 0xca0,
0x460, 0x569, 0x663, 0x76a, 0x66 , 0x16f, 0x265, 0x36c,
0xc6c, 0xd65, 0xe6f, 0xf66, 0x86a, 0x963, 0xa69, 0xb60,
0x5f0, 0x4f9, 0x7f3, 0x6fa, 0x1f6, 0xff , 0x3f5, 0x2fc,
0xdfc, 0xcf5, 0xfff, 0xef6, 0x9fa, 0x8f3, 0xbf9, 0xaf0,
0x650, 0x759, 0x453, 0x55a, 0x256, 0x35f, 0x55 , 0x15c,
0xe5c, 0xf55, 0xc5f, 0xd56, 0xa5a, 0xb53, 0x859, 0x950,
0x7c0, 0x6c9, 0x5c3, 0x4ca, 0x3c6, 0x2cf, 0x1c5, 0xcc ,
0xfcc, 0xec5, 0xdcf, 0xcc6, 0xbca, 0xac3, 0x9c9, 0x8c0,
0x8c0, 0x9c9, 0xac3, 0xbca, 0xcc6, 0xdcf, 0xec5, 0xfcc,
0xcc , 0x1c5, 0x2cf, 0x3c6, 0x4ca, 0x5c3, 0x6c9, 0x7c0,
0x950, 0x859, 0xb53, 0xa5a, 0xd56, 0xc5f, 0xf55, 0xe5c,
0x15c, 0x55 , 0x35f, 0x256, 0x55a, 0x453, 0x759, 0x650,
0xaf0, 0xbf9, 0x8f3, 0x9fa, 0xef6, 0xfff, 0xcf5, 0xdfc,
0x2fc, 0x3f5, 0xff , 0x1f6, 0x6fa, 0x7f3, 0x4f9, 0x5f0,
0xb60, 0xa69, 0x963, 0x86a, 0xf66, 0xe6f, 0xd65, 0xc6c,
0x36c, 0x265, 0x16f, 0x66 , 0x76a, 0x663, 0x569, 0x460,
0xca0, 0xda9, 0xea3, 0xfaa, 0x8a6, 0x9af, 0xaa5, 0xbac,
0x4ac, 0x5a5, 0x6af, 0x7a6, 0xaa , 0x1a3, 0x2a9, 0x3a0,
0xd30, 0xc39, 0xf33, 0xe3a, 0x936, 0x83f, 0xb35, 0xa3c,
0x53c, 0x435, 0x73f, 0x636, 0x13a, 0x33 , 0x339, 0x230,
0xe90, 0xf99, 0xc93, 0xd9a, 0xa96, 0xb9f, 0x895, 0x99c,
0x69c, 0x795, 0x49f, 0x596, 0x29a, 0x393, 0x99 , 0x190,
0xf00, 0xe09, 0xd03, 0xc0a, 0xb06, 0xa0f, 0x905, 0x80c,
0x70c, 0x605, 0x50f, 0x406, 0x30a, 0x203, 0x109, 0x0   };
int triTable[256][16] =
{{−1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 3, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 1, 9, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 8, 3, 9, 8, 1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 10, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 3, 1, 2, 10, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{9, 2, 10, 0, 2, 9, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{2, 8, 3, 2, 10, 8, 10, 9, 8, −1, −1, −1, −1, −1, −1, −1},
{3, 11, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 11, 2, 8, 11, 0, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 9, 0, 2, 3, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 11, 2, 1, 9, 11, 9, 8, 11, −1, −1, −1, −1, −1, −1, −1},
{3, 10, 1, 11, 10, 3, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 10, 1, 0, 8, 10, 8, 11, 10, −1, −1, −1, −1, −1, −1, −1},
{3, 9, 0, 3, 11, 9, 11, 10, 9, −1, −1, −1, −1, −1, −1, −1},
{9, 8, 10, 10, 8, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 7, 8, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 3, 0, 7, 3, 4, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 1, 9, 8, 4, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 1, 9, 4, 7, 1, 7, 3, 1, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 10, 8, 4, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{3, 4, 7, 3, 0, 4, 1, 2, 10, −1, −1, −1, −1, −1, −1, −1},
{9, 2, 10, 9, 0, 2, 8, 4, 7, −1, −1, −1, −1, −1, −1, −1},
{2, 10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, −1, −1, −1, −1},
{8, 4, 7, 3, 11, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{11, 4, 7, 11, 2, 4, 2, 0, 4, −1, −1, −1, −1, −1, −1, −1},
{9, 0, 1, 8, 4, 7, 2, 3, 11, −1, −1, −1, −1, −1, −1, −1},
{4, 7, 11, 9, 4, 11, 9, 11, 2, 9, 2, 1, −1, −1, −1, −1},
{3, 10, 1, 3, 11, 10, 7, 8, 4, −1, −1, −1, −1, −1, −1, −1},
{1, 11, 10, 1, 4, 11, 1, 0, 4, 7, 11, 4, −1, −1, −1, −1},
{4, 7, 8, 9, 0, 11, 9, 11, 10, 11, 0, 3, −1, −1, −1, −1},
{4, 7, 11, 4, 11, 9, 9, 11, 10, −1, −1, −1, −1, −1, −1, −1},
{9, 5, 4, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{9, 5, 4, 0, 8, 3, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 5, 4, 1, 5, 0, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{8, 5, 4, 8, 3, 5, 3, 1, 5, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 10, 9, 5, 4, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{3, 0, 8, 1, 2, 10, 4, 9, 5, −1, −1, −1, −1, −1, −1, −1},
{5, 2, 10, 5, 4, 2, 4, 0, 2, −1, −1, −1, −1, −1, −1, −1},
{2, 10, 5, 3, 2, 5, 3, 5, 4, 3, 4, 8, −1, −1, −1, −1},
{9, 5, 4, 2, 3, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 11, 2, 0, 8, 11, 4, 9, 5, −1, −1, −1, −1, −1, −1, −1},
{0, 5, 4, 0, 1, 5, 2, 3, 11, −1, −1, −1, −1, −1, −1, −1},
{2, 1, 5, 2, 5, 8, 2, 8, 11, 4, 8, 5, −1, −1, −1, −1},
{10, 3, 11, 10, 1, 3, 9, 5, 4, −1, −1, −1, −1, −1, −1, −1},
{4, 9, 5, 0, 8, 1, 8, 10, 1, 8, 11, 10, −1, −1, −1, −1},
{5, 4, 0, 5, 0, 11, 5, 11, 10, 11, 0, 3, −1, −1, −1, −1},
{5, 4, 8, 5, 8, 10, 10, 8, 11, −1, −1, −1, −1, −1, −1, −1},
{9, 7, 8, 5, 7, 9, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{9, 3, 0, 9, 5, 3, 5, 7, 3, −1, −1, −1, −1, −1, −1, −1},
{0, 7, 8, 0, 1, 7, 1, 5, 7, −1, −1, −1, −1, −1, −1, −1},
{1, 5, 3, 3, 5, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{9, 7, 8, 9, 5, 7, 10, 1, 2, −1, −1, −1, −1, −1, −1, −1},
{10, 1, 2, 9, 5, 0, 5, 3, 0, 5, 7, 3, −1, −1, −1, −1},
{8, 0, 2, 8, 2, 5, 8, 5, 7, 10, 5, 2, −1, −1, −1, −1},
{2, 10, 5, 2, 5, 3, 3, 5, 7, −1, −1, −1, −1, −1, −1, −1},
{7, 9, 5, 7, 8, 9, 3, 11, 2, −1, −1, −1, −1, −1, −1, −1},
{9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7, 11, −1, −1, −1, −1},
{2, 3, 11, 0, 1, 8, 1, 7, 8, 1, 5, 7, −1, −1, −1, −1},
{11, 2, 1, 11, 1, 7, 7, 1, 5, −1, −1, −1, −1, −1, −1, −1},
{9, 5, 8, 8, 5, 7, 10, 1, 3, 10, 3, 11, −1, −1, −1, −1},
{5, 7, 0, 5, 0, 9, 7, 11, 0, 1, 0, 10, 11, 10, 0, −1},
{11, 10, 0, 11, 0, 3, 10, 5, 0, 8, 0, 7, 5, 7, 0, −1},
{11, 10, 5, 7, 11, 5, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{10, 6, 5, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 3, 5, 10, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{9, 0, 1, 5, 10, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 8, 3, 1, 9, 8, 5, 10, 6, −1, −1, −1, −1, −1, −1, −1},
{1, 6, 5, 2, 6, 1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 6, 5, 1, 2, 6, 3, 0, 8, −1, −1, −1, −1, −1, −1, −1},
{9, 6, 5, 9, 0, 6, 0, 2, 6, −1, −1, −1, −1, −1, −1, −1},
{5, 9, 8, 5, 8, 2, 5, 2, 6, 3, 2, 8, −1, −1, −1, −1},
{2, 3, 11, 10, 6, 5, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{11, 0, 8, 11, 2, 0, 10, 6, 5, −1, −1, −1, −1, −1, −1, −1},
{0, 1, 9, 2, 3, 11, 5, 10, 6, −1, −1, −1, −1, −1, −1, −1},
{5, 10, 6, 1, 9, 2, 9, 11, 2, 9, 8, 11, −1, −1, −1, −1},
{6, 3, 11, 6, 5, 3, 5, 1, 3, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 11, 0, 11, 5, 0, 5, 1, 5, 11, 6, −1, −1, −1, −1},
{3, 11, 6, 0, 3, 6, 0, 6, 5, 0, 5, 9, −1, −1, −1, −1},
{6, 5, 9, 6, 9, 11, 11, 9, 8, −1, −1, −1, −1, −1, −1, −1},
{5, 10, 6, 4, 7, 8, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 3, 0, 4, 7, 3, 6, 5, 10, −1, −1, −1, −1, −1, −1, −1},
{1, 9, 0, 5, 10, 6, 8, 4, 7, −1, −1, −1, −1, −1, −1, −1},
{10, 6, 5, 1, 9, 7, 1, 7, 3, 7, 9, 4, −1, −1, −1, −1},
{6, 1, 2, 6, 5, 1, 4, 7, 8, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 5, 5, 2, 6, 3, 0, 4, 3, 4, 7, −1, −1, −1, −1},
{8, 4, 7, 9, 0, 5, 0, 6, 5, 0, 2, 6, −1, −1, −1, −1},
{7, 3, 9, 7, 9, 4, 3, 2, 9, 5, 9, 6, 2, 6, 9, −1},
{3, 11, 2, 7, 8, 4, 10, 6, 5, −1, −1, −1, −1, −1, −1, −1},
{5, 10, 6, 4, 7, 2, 4, 2, 0, 2, 7, 11, −1, −1, −1, −1},
{0, 1, 9, 4, 7, 8, 2, 3, 11, 5, 10, 6, −1, −1, −1, −1},
{9, 2, 1, 9, 11, 2, 9, 4, 11, 7, 11, 4, 5, 10, 6, −1},
{8, 4, 7, 3, 11, 5, 3, 5, 1, 5, 11, 6, −1, −1, −1, −1},
{5, 1, 11, 5, 11, 6, 1, 0, 11, 7, 11, 4, 0, 4, 11, −1},
{0, 5, 9, 0, 6, 5, 0, 3, 6, 11, 6, 3, 8, 4, 7, −1},
{6, 5, 9, 6, 9, 11, 4, 7, 9, 7, 11, 9, −1, −1, −1, −1},
{10, 4, 9, 6, 4, 10, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 10, 6, 4, 9, 10, 0, 8, 3, −1, −1, −1, −1, −1, −1, −1},
{10, 0, 1, 10, 6, 0, 6, 4, 0, −1, −1, −1, −1, −1, −1, −1},
{8, 3, 1, 8, 1, 6, 8, 6, 4, 6, 1, 10, −1, −1, −1, −1},
{1, 4, 9, 1, 2, 4, 2, 6, 4, −1, −1, −1, −1, −1, −1, −1},
{3, 0, 8, 1, 2, 9, 2, 4, 9, 2, 6, 4, −1, −1, −1, −1},
{0, 2, 4, 4, 2, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{8, 3, 2, 8, 2, 4, 4, 2, 6, −1, −1, −1, −1, −1, −1, −1},
{10, 4, 9, 10, 6, 4, 11, 2, 3, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 2, 2, 8, 11, 4, 9, 10, 4, 10, 6, −1, −1, −1, −1},
{3, 11, 2, 0, 1, 6, 0, 6, 4, 6, 1, 10, −1, −1, −1, −1},
{6, 4, 1, 6, 1, 10, 4, 8, 1, 2, 1, 11, 8, 11, 1, −1},
{9, 6, 4, 9, 3, 6, 9, 1, 3, 11, 6, 3, −1, −1, −1, −1},
{8, 11, 1, 8, 1, 0, 11, 6, 1, 9, 1, 4, 6, 4, 1, −1},
{3, 11, 6, 3, 6, 0, 0, 6, 4, −1, −1, −1, −1, −1, −1, −1},
{6, 4, 8, 11, 6, 8, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{7, 10, 6, 7, 8, 10, 8, 9, 10, −1, −1, −1, −1, −1, −1, −1},
{0, 7, 3, 0, 10, 7, 0, 9, 10, 6, 7, 10, −1, −1, −1, −1},
{10, 6, 7, 1, 10, 7, 1, 7, 8, 1, 8, 0, −1, −1, −1, −1},
{10, 6, 7, 10, 7, 1, 1, 7, 3, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 6, 1, 6, 8, 1, 8, 9, 8, 6, 7, −1, −1, −1, −1},
{2, 6, 9, 2, 9, 1, 6, 7, 9, 0, 9, 3, 7, 3, 9, −1},
{7, 8, 0, 7, 0, 6, 6, 0, 2, −1, −1, −1, −1, −1, −1, −1},
{7, 3, 2, 6, 7, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{2, 3, 11, 10, 6, 8, 10, 8, 9, 8, 6, 7, −1, −1, −1, −1},
{2, 0, 7, 2, 7, 11, 0, 9, 7, 6, 7, 10, 9, 10, 7, −1},
{1, 8, 0, 1, 7, 8, 1, 10, 7, 6, 7, 10, 2, 3, 11, −1},
{11, 2, 1, 11, 1, 7, 10, 6, 1, 6, 7, 1, −1, −1, −1, −1},
{8, 9, 6, 8, 6, 7, 9, 1, 6, 11, 6, 3, 1, 3, 6, −1},
{0, 9, 1, 11, 6, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{7, 8, 0, 7, 0, 6, 3, 11, 0, 11, 6, 0, −1, −1, −1, −1},
{7, 11, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{7, 6, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{3, 0, 8, 11, 7, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 1, 9, 11, 7, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{8, 1, 9, 8, 3, 1, 11, 7, 6, −1, −1, −1, −1, −1, −1, −1},
{10, 1, 2, 6, 11, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
```

```
{1, 2, 10, 3, 0, 8, 6, 11, 7, −1, −1, −1, −1, −1, −1, −1},
{2, 9, 0, 2, 10, 9, 6, 11, 7, −1, −1, −1, −1, −1, −1, −1},
{6, 11, 7, 2, 10, 3, 10, 8, 3, 10, 9, 8, −1, −1, −1, −1},
{7, 2, 3, 6, 2, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{7, 0, 8, 7, 6, 0, 6, 2, 0, −1, −1, −1, −1, −1, −1, −1},
{2, 7, 6, 2, 3, 7, 0, 1, 9, −1, −1, −1, −1, −1, −1, −1},
{1, 6, 2, 1, 8, 6, 1, 9, 8, 8, 7, 6, −1, −1, −1, −1},
{10, 7, 6, 10, 1, 7, 1, 3, 7, −1, −1, −1, −1, −1, −1, −1},
{10, 7, 6, 1, 7, 10, 1, 8, 7, 1, 0, 8, −1, −1, −1, −1},
{0, 3, 7, 0, 7, 10, 0, 10, 9, 6, 10, 7, −1, −1, −1, −1},
{7, 6, 10, 7, 10, 8, 8, 10, 9, −1, −1, −1, −1, −1, −1, −1},
{6, 8, 4, 11, 8, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{3, 6, 11, 3, 0, 6, 0, 4, 6, −1, −1, −1, −1, −1, −1, −1},
{8, 6, 11, 8, 4, 6, 9, 0, 1, −1, −1, −1, −1, −1, −1, −1},
{9, 4, 6, 9, 6, 3, 9, 3, 1, 11, 3, 6, −1, −1, −1, −1},
{6, 8, 4, 6, 11, 8, 2, 10, 1, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 10, 3, 0, 11, 0, 6, 11, 0, 4, 6, −1, −1, −1, −1},
{4, 11, 8, 4, 6, 11, 0, 2, 9, 2, 10, 9, −1, −1, −1, −1},
{10, 9, 3, 10, 3, 2, 9, 4, 3, 11, 3, 6, 4, 6, 3, −1},
{8, 2, 3, 8, 4, 2, 4, 6, 2, −1, −1, −1, −1, −1, −1, −1},
{0, 4, 2, 4, 6, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 9, 0, 2, 3, 4, 2, 4, 6, 4, 3, 8, −1, −1, −1, −1},
{1, 9, 4, 1, 4, 2, 2, 4, 6, −1, −1, −1, −1, −1, −1, −1},
{8, 1, 3, 8, 6, 1, 8, 4, 6, 6, 10, 1, −1, −1, −1, −1},
{10, 1, 0, 10, 0, 6, 6, 0, 4, −1, −1, −1, −1, −1, −1, −1},
{4, 6, 3, 4, 3, 8, 6, 10, 3, 0, 3, 9, 10, 9, 3, −1},
{10, 9, 4, 6, 10, 4, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 9, 5, 7, 6, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 3, 4, 9, 5, 11, 7, 6, −1, −1, −1, −1, −1, −1, −1},
{5, 0, 1, 5, 4, 0, 7, 6, 11, −1, −1, −1, −1, −1, −1, −1},
{11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5, −1, −1, −1, −1},
{9, 5, 4, 10, 1, 2, 7, 6, 11, −1, −1, −1, −1, −1, −1, −1},
{6, 11, 7, 1, 2, 10, 0, 8, 3, 4, 9, 5, −1, −1, −1, −1},
{7, 6, 11, 5, 4, 10, 4, 2, 10, 4, 0, 2, −1, −1, −1, −1},
{3, 4, 8, 3, 5, 4, 3, 2, 5, 10, 5, 2, 11, 7, 6, −1},
{7, 2, 3, 7, 6, 2, 5, 4, 9, −1, −1, −1, −1, −1, −1, −1},
{9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7, −1, −1, −1, −1},
{3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0, −1, −1, −1, −1},
{6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8, −1},
{9, 5, 4, 10, 1, 6, 1, 7, 6, 1, 3, 7, −1, −1, −1, −1},
{1, 6, 10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4, −1},
{4, 0, 10, 4, 10, 5, 0, 3, 10, 6, 10, 7, 3, 7, 10, −1},
{7, 6, 10, 7, 10, 8, 5, 4, 10, 4, 8, 10, −1, −1, −1, −1},
{6, 9, 5, 6, 11, 9, 11, 8, 9, −1, −1, −1, −1, −1, −1, −1},
{3, 6, 11, 0, 6, 3, 0, 5, 6, 0, 9, 5, −1, −1, −1, −1},
{0, 11, 8, 0, 5, 11, 0, 1, 5, 5, 6, 11, −1, −1, −1, −1},
{6, 11, 3, 6, 3, 5, 5, 3, 1, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 10, 9, 5, 11, 9, 11, 8, 11, 5, 6, −1, −1, −1, −1},
{0, 11, 3, 0, 6, 11, 0, 9, 6, 5, 6, 9, 1, 2, 10, −1},
{11, 8, 5, 11, 5, 6, 8, 0, 5, 10, 5, 2, 0, 2, 5, −1},
{6, 11, 3, 6, 3, 5, 2, 10, 3, 10, 5, 3, −1, −1, −1, −1},
{5, 8, 9, 5, 2, 8, 5, 6, 2, 3, 8, 2, −1, −1, −1, −1},
{9, 5, 6, 9, 6, 0, 0, 6, 2, −1, −1, −1, −1, −1, −1, −1},
{1, 5, 8, 1, 8, 0, 5, 6, 8, 3, 8, 2, 6, 2, 8, −1},
{1, 5, 6, 2, 1, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 3, 6, 1, 6, 10, 3, 8, 6, 5, 6, 9, 8, 9, 6, −1},
{10, 1, 0, 10, 0, 6, 9, 5, 0, 5, 6, 0, −1, −1, −1, −1},
{0, 3, 8, 5, 6, 10, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{10, 5, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{11, 5, 10, 7, 5, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{11, 5, 10, 11, 7, 5, 8, 3, 0, −1, −1, −1, −1, −1, −1, −1},
{5, 11, 7, 5, 10, 11, 1, 9, 0, −1, −1, −1, −1, −1, −1, −1},
{10, 7, 5, 10, 11, 7, 9, 8, 1, 8, 3, 1, −1, −1, −1, −1},
{11, 1, 2, 11, 7, 1, 7, 5, 1, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 3, 1, 2, 7, 1, 7, 5, 7, 2, 11, −1, −1, −1, −1},
{9, 7, 5, 9, 2, 7, 9, 0, 2, 2, 11, 7, −1, −1, −1, −1},
{7, 5, 2, 7, 2, 11, 5, 9, 2, 3, 2, 8, 9, 8, 2, −1},
{2, 5, 10, 2, 3, 5, 3, 7, 5, −1, −1, −1, −1, −1, −1, −1},
{8, 2, 0, 8, 5, 2, 8, 7, 5, 10, 2, 5, −1, −1, −1, −1},
{9, 0, 1, 5, 10, 3, 5, 3, 7, 3, 10, 2, −1, −1, −1, −1},
{9, 8, 2, 9, 2, 1, 8, 7, 2, 10, 2, 5, 7, 5, 2, −1},
{1, 3, 5, 3, 7, 5, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 7, 0, 7, 1, 1, 7, 5, −1, −1, −1, −1, −1, −1, −1},
{9, 0, 3, 9, 3, 5, 5, 3, 7, −1, −1, −1, −1, −1, −1, −1},
{9, 8, 7, 5, 9, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{5, 8, 4, 5, 10, 8, 10, 11, 8, −1, −1, −1, −1, −1, −1, −1},
{5, 0, 4, 5, 11, 0, 5, 10, 11, 11, 3, 0, −1, −1, −1, −1},
{0, 1, 9, 8, 4, 10, 8, 10, 11, 10, 4, 5, −1, −1, −1, −1},
{10, 11, 4, 10, 4, 5, 11, 3, 4, 9, 4, 1, 3, 1, 4, −1},
{2, 5, 1, 2, 8, 5, 2, 11, 8, 4, 5, 8, −1, −1, −1, −1},
{0, 4, 11, 0, 11, 3, 4, 5, 11, 2, 11, 1, 5, 1, 11, −1},
{0, 2, 5, 0, 5, 9, 2, 11, 5, 4, 5, 8, 11, 8, 5, −1},
{9, 4, 5, 2, 11, 3, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{2, 5, 10, 3, 5, 2, 3, 4, 5, 3, 8, 4, −1, −1, −1, −1},
{5, 10, 2, 5, 2, 4, 4, 2, 0, −1, −1, −1, −1, −1, −1, −1},
{3, 10, 2, 3, 5, 10, 3, 8, 5, 4, 5, 8, 0, 1, 9, −1},
{5, 10, 2, 5, 2, 4, 1, 9, 2, 9, 4, 2, −1, −1, −1, −1},
{8, 4, 5, 8, 5, 3, 3, 5, 1, −1, −1, −1, −1, −1, −1, −1},
{0, 4, 5, 1, 0, 5, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{8, 4, 5, 8, 5, 3, 9, 0, 5, 0, 3, 5, −1, −1, −1, −1},
{9, 4, 5, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 11, 7, 4, 9, 11, 9, 10, 11, −1, −1, −1, −1, −1, −1, −1},
{0, 8, 3, 4, 9, 7, 9, 11, 7, 9, 10, 11, −1, −1, −1, −1},
{1, 10, 11, 1, 11, 4, 1, 4, 0, 7, 4, 11, −1, −1, −1, −1},
{3, 1, 4, 3, 4, 8, 1, 10, 4, 7, 4, 11, 10, 11, 4, −1},
{4, 11, 7, 9, 11, 4, 9, 2, 11, 9, 1, 2, −1, −1, −1, −1},
{9, 7, 4, 9, 11, 7, 9, 1, 11, 2, 11, 1, 0, 8, 3, −1},
{11, 7, 4, 11, 4, 2, 2, 4, 0, −1, −1, −1, −1, −1, −1, −1},
{11, 7, 4, 11, 4, 2, 8, 3, 4, 3, 2, 4, −1, −1, −1, −1},
{2, 9, 10, 2, 7, 9, 2, 3, 7, 7, 4, 9, −1, −1, −1, −1},
{9, 10, 7, 9, 7, 4, 10, 2, 7, 8, 7, 0, 2, 0, 7, −1},
{3, 7, 10, 3, 10, 2, 7, 4, 10, 1, 10, 0, 4, 0, 10, −1},
{1, 10, 2, 8, 7, 4, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 9, 1, 4, 1, 7, 7, 1, 3, −1, −1, −1, −1, −1, −1, −1},
{4, 9, 1, 4, 1, 7, 0, 8, 1, 8, 7, 1, −1, −1, −1, −1},
{4, 0, 3, 7, 4, 3, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{4, 8, 7, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{9, 10, 8, 10, 11, 8, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{3, 0, 9, 3, 9, 11, 11, 9, 10, −1, −1, −1, −1, −1, −1, −1},
{0, 1, 10, 0, 10, 8, 8, 10, 11, −1, −1, −1, −1, −1, −1, −1},
{3, 1, 10, 11, 3, 10, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 2, 11, 1, 11, 9, 9, 11, 8, −1, −1, −1, −1, −1, −1, −1},
{3, 0, 9, 3, 9, 11, 1, 2, 9, 2, 11, 9, −1, −1, −1, −1},
{0, 2, 11, 8, 0, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{3, 2, 11, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{2, 3, 8, 2, 8, 10, 10, 8, 9, −1, −1, −1, −1, −1, −1, −1},
{9, 10, 2, 0, 9, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{2, 3, 8, 2, 8, 10, 0, 1, 8, 1, 10, 8, −1, −1, −1, −1},
{1, 10, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{1, 3, 8, 9, 1, 8, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 9, 1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{0, 3, 8, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1},
{−1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1}};

/*
   Determine the index into the edge table which
   tells us which vertices are inside of the surface
*/
cubeindex = 0;
if (grid.val[0] < isolevel) cubeindex |= 1;
if (grid.val[1] < isolevel) cubeindex |= 2;
if (grid.val[2] < isolevel) cubeindex |= 4;
if (grid.val[3] < isolevel) cubeindex |= 8;
if (grid.val[4] < isolevel) cubeindex |= 16;
if (grid.val[5] < isolevel) cubeindex |= 32;
if (grid.val[6] < isolevel) cubeindex |= 64;
if (grid.val[7] < isolevel) cubeindex |= 128;

/* Cube is entirely in/out of the surface */
if (edgeTable[cubeindex] == 0)
  return(0);

/* Find the vertices where the surface intersects the cube */
if (edgeTable[cubeindex] & 1)
  vertlist[0] =
     VertexInterp(isolevel,grid.p[0],grid.p[1],grid.val[0],grid.val[1]);
if (edgeTable[cubeindex] & 2)
  vertlist[1] =
     VertexInterp(isolevel,grid.p[1],grid.p[2],grid.val[1],grid.val[2]);
if (edgeTable[cubeindex] & 4)
  vertlist[2] =
     VertexInterp(isolevel,grid.p[2],grid.p[3],grid.val[2],grid.val[3]);
if (edgeTable[cubeindex] & 8)
  vertlist[3] =
     VertexInterp(isolevel,grid.p[3],grid.p[0],grid.val[3],grid.val[0]);
if (edgeTable[cubeindex] & 16)
  vertlist[4] =
     VertexInterp(isolevel,grid.p[4],grid.p[5],grid.val[4],grid.val[5]);
if (edgeTable[cubeindex] & 32)
  vertlist[5] =
     VertexInterp(isolevel,grid.p[5],grid.p[6],grid.val[5],grid.val[6]);
if (edgeTable[cubeindex] & 64)
  vertlist[6] =
     VertexInterp(isolevel,grid.p[6],grid.p[7],grid.val[6],grid.val[7]);
if (edgeTable[cubeindex] & 128)
  vertlist[7] =
     VertexInterp(isolevel,grid.p[7],grid.p[4],grid.val[7],grid.val[4]);
if (edgeTable[cubeindex] & 256)
  vertlist[8] =
     VertexInterp(isolevel,grid.p[0],grid.p[4],grid.val[0],grid.val[4]);
if (edgeTable[cubeindex] & 512)
  vertlist[9] =
     VertexInterp(isolevel,grid.p[1],grid.p[5],grid.val[1],grid.val[5]);
if (edgeTable[cubeindex] & 1024)
  vertlist[10] =
     VertexInterp(isolevel,grid.p[2],grid.p[6],grid.val[2],grid.val[6]);
if (edgeTable[cubeindex] & 2048)
  vertlist[11] =
     VertexInterp(isolevel,grid.p[3],grid.p[7],grid.val[3],grid.val[7]);
```

```
371     /* Create the triangle */
372     ntriang = 0;
373     for (i=0;triTable[cubeindex][i]!=-1;i+=3) {
374       triangles[ntriang].p[0] = vertlist[triTable[cubeindex][i  ]];
375       triangles[ntriang].p[1] = vertlist[triTable[cubeindex][i+1]];
376       triangles[ntriang].p[2] = vertlist[triTable[cubeindex][i+2]];
377       ntriang++;
378     }
379
380     return(ntriang);
381   }
382
383   /*
384     Linearly interpolate the position where an isosurface cuts
385     an edge between two vertices, each with their own scalar value
386   */
387   XYZ VertexInterp(isolevel,p1,p2,valp1,valp2)
388   double isolevel;
389   XYZ p1,p2;
390   double valp1,valp2;
391   {
392     double mu;
393     XYZ p;
394
395     if (ABS(isolevel-valp1) < 0.00001)
396       return(p1);
397     if (ABS(isolevel-valp2) < 0.00001)
398       return(p2);
399     if (ABS(valp1-valp2) < 0.00001)
400       return(p1);
401     mu = (isolevel - valp1) / (valp2 - valp1);
402     p.x = p1.x + mu * (p2.x - p1.x);
403     p.y = p1.y + mu * (p2.y - p1.y);
404     p.z = p1.z + mu * (p2.z - p1.z);
405
406     return(p);
407   }
```