CS3388B: Lecture 12

March 14, 2023

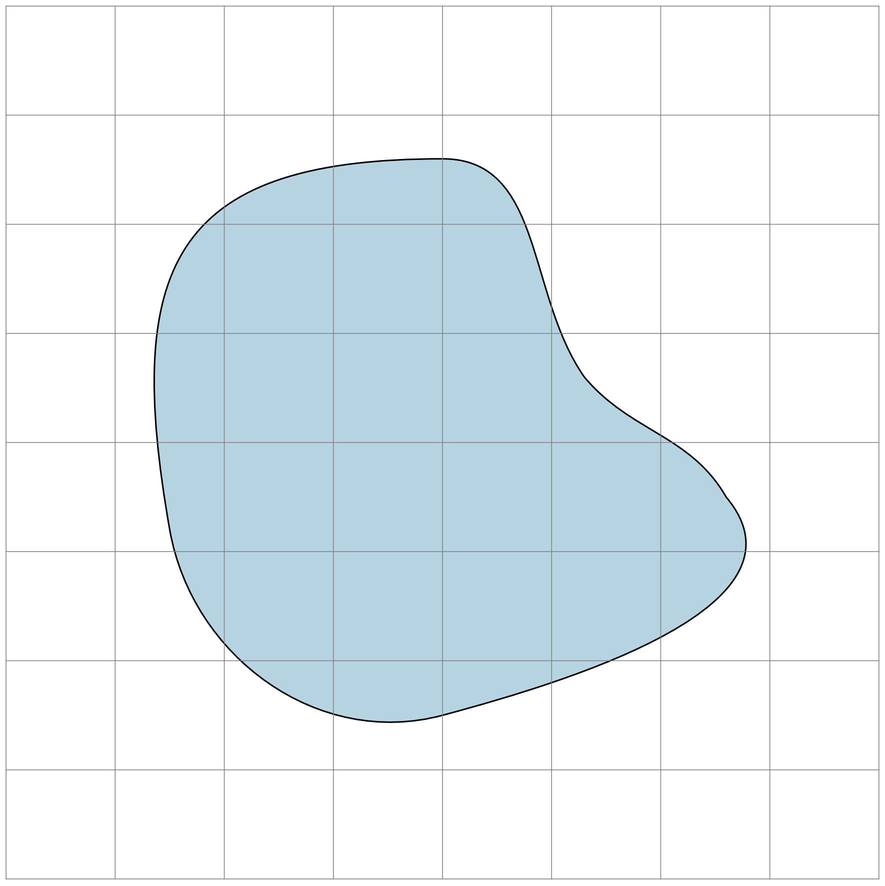
12 Marching Cubes

Now we know how to render and store triangle meshes. In this section we will look at how to create triangle meshes. This algorithm is known as **Marching Cubes**. The algorithm follows a simple idea: search through space to find which areas of inside a surface or outside a surface. Then, draw a surface in between what is outside and what is inside.

12.1 Marching Squares

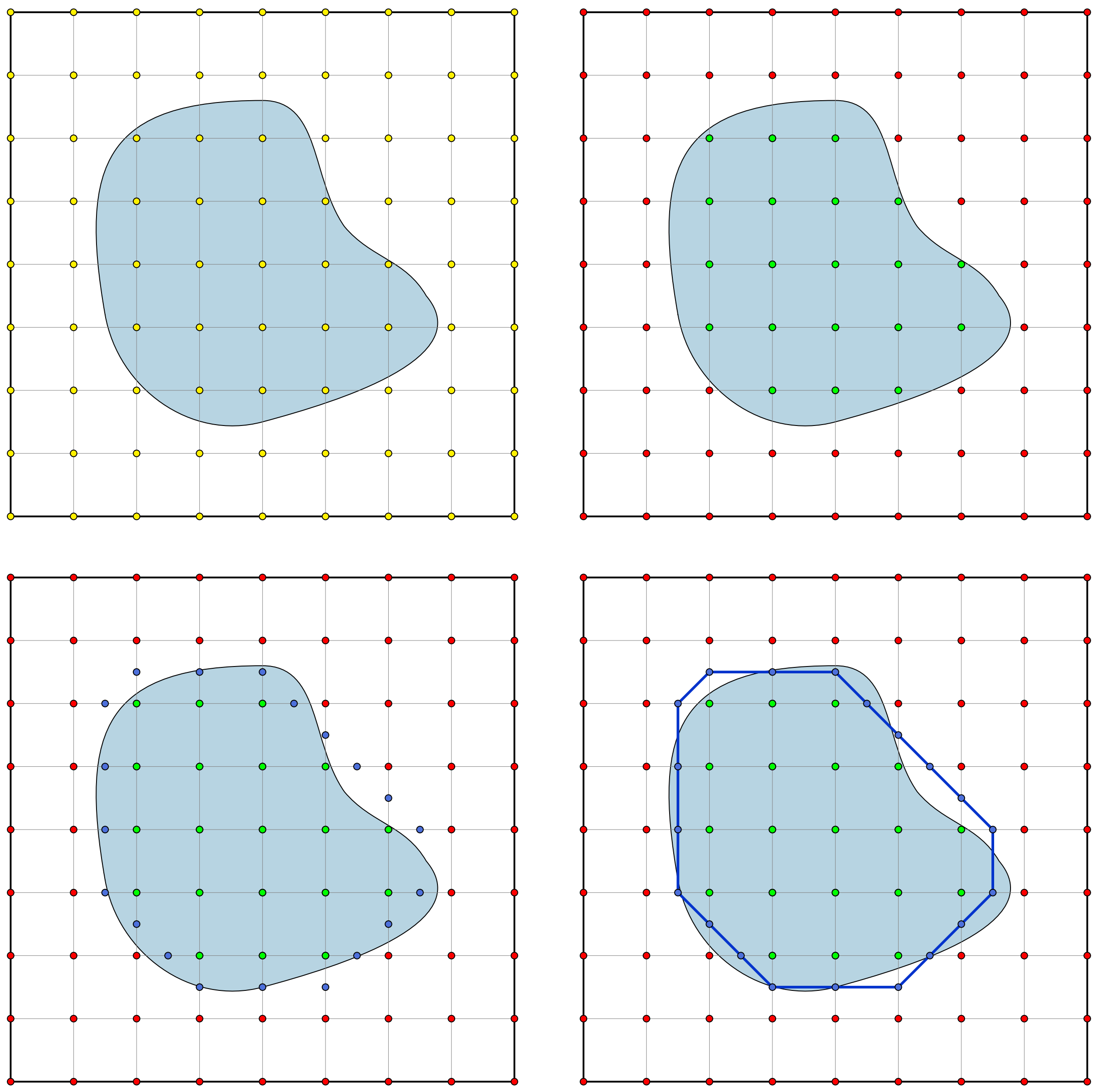
Let's begin by looking at a two-dimensional example to get the idea. Consider an arbitrary object in two dimensions. Drawing arbitrary curves in computer graphics is quite challenging due to discrete nature of pixels, for example. Moreover, sometimes we only have a mathematical description of an object.

In either case, we want to compute a triangular mesh to approximate this object and then use that mesh to render th e object. This is the goal of marching cubes.



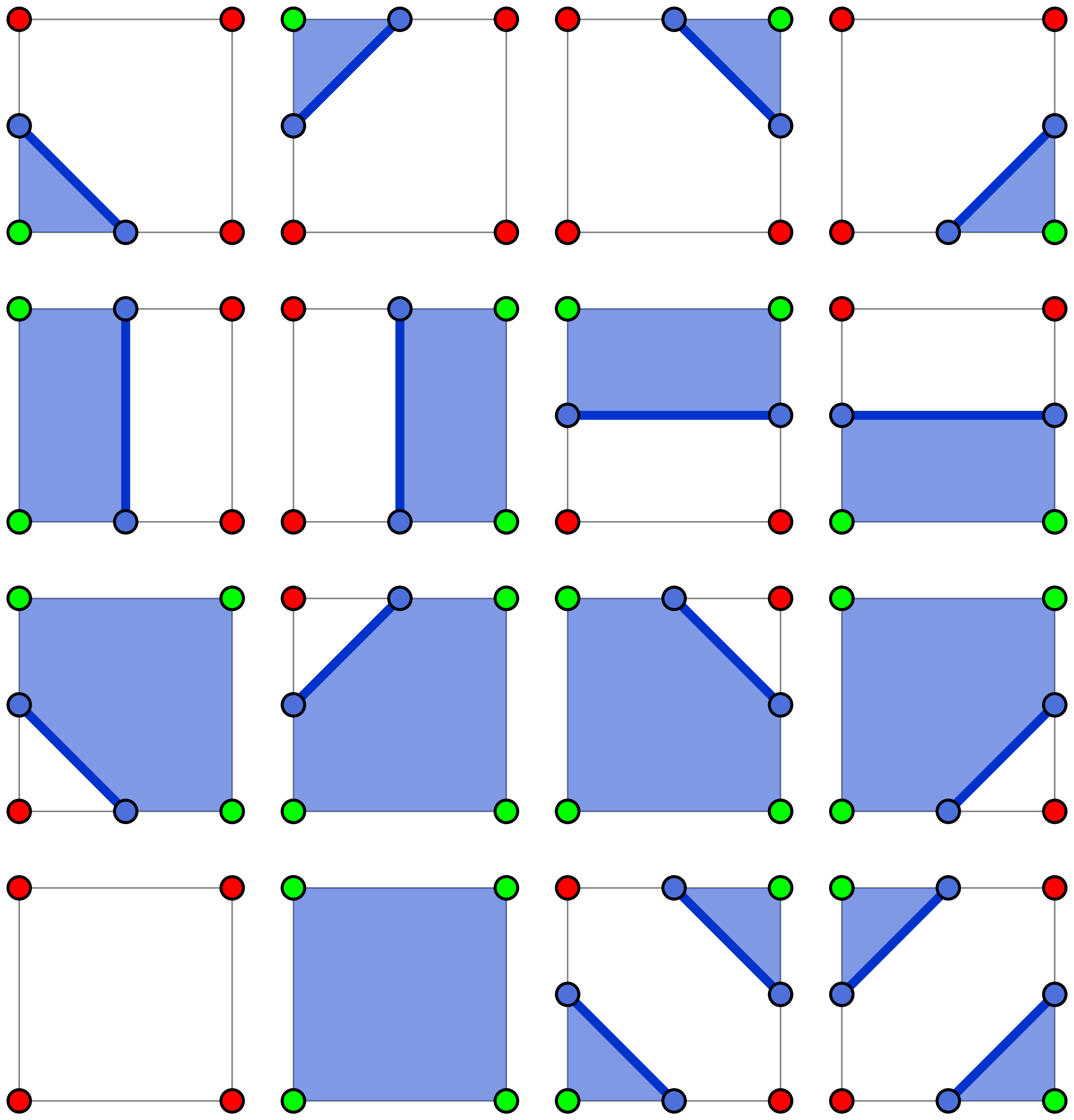
Marching Squares proceeds in 4 steps:

1. **Divide** the area (domain) on which the object is described into a regular grid of squares.
2. For each square in the grid **test** whether each of the corners of that square are **inside** or **outside** the object.
3. **Generate** a vertex along any edge that has one corner inside the surface and one corner outside the surface,
4. **Connect** each generated vertex to its nearest neighbours.



In fact, steps 3 and 4 can even be further simplified. For each square, which corners are inside or outside the surface falls into one of 16 categories of line segments.

Therefore, one can have a predefined list of line segments in a **look up table** which are chosen based on which corners of the square are inside the object and which are outside.



12.2 Bitmaps for Lookup Tables

A **bitmap** is a simple structure which uses the individual bits of binary number to encode several different Boolean values simultaneously. Each bit of the number represents one particular choice of Boolean value: true or false. Then, the combination of Boolean values can be considered as a number to handle compound cases.

Consider the case of marching squares. There are four corners to each square, and each corner can either be inside or outside the object. Thus, there are four Boolean values which are independently true or false. Let us consider **inside as true** and **outside as false**.

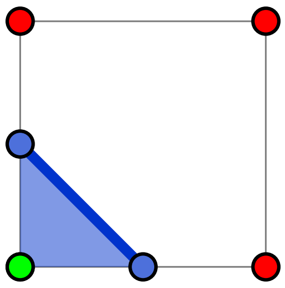
Consider a 4-bit binary number. Let the most significant bit correspond to the top-left corner, the second-most significant bit be the top-right corner, the second-least significant bit be the bottom-right corner, and the least significant bit be the bottom-left corner. Then, all cases for the marching squares algorithm can be encoded as the numbers 0 to 15.

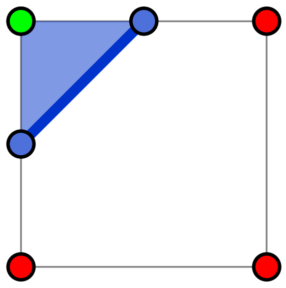
The key to bitmaps is **bit-wise operations** and **bitmasks**. The correspondence between corners and binary digits in the previous paragraph leads to the following bit masks:

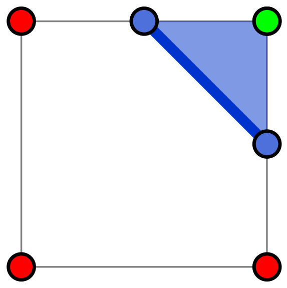
Top-left corner:	`1000`	Top-right corner:	`0100`
Bottom-right corner:	`0010`	Bottom-left corner:	`0001`

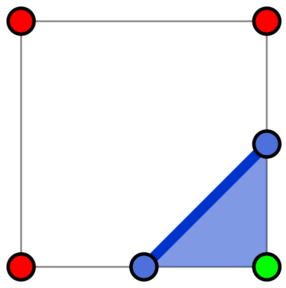
For a binary number encoding multiple a particular combined case of the four Booleans, we can extract individual Boolean values using these masks. Simply use **bitwise AND** between the binary number and mask corresponding to the case you want to check. If you want to check if the top-left corner is “true”.

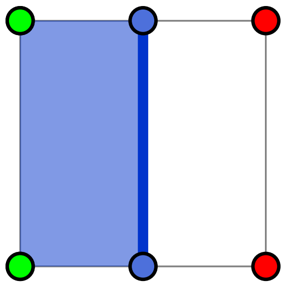
```
#define TOP_LEFT     8
#define TOP_RIGHT    4
#define BOTTOM_RIGHT 2
#define BOTTOM_LEFT  1

int case = 14;
if (case & TOP_LEFT) {
    //This case has the top-left corner inside the object
}
if (case & TOP_RIGHT) {
    //This case has the top-right corner inside the object
}
//...
```

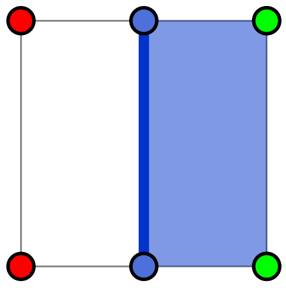

0001 = 1

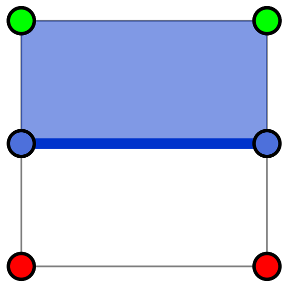

1000 = 8

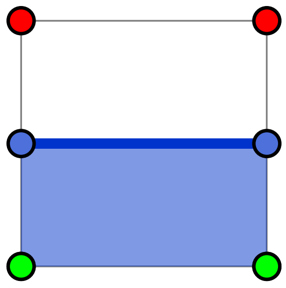

0100 = 4

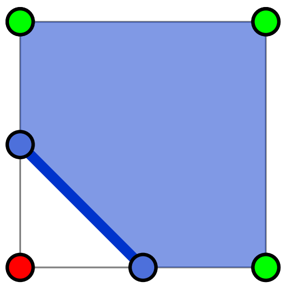

0010 = 2

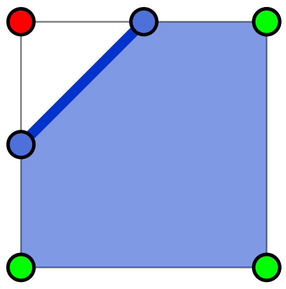

1001 = 9

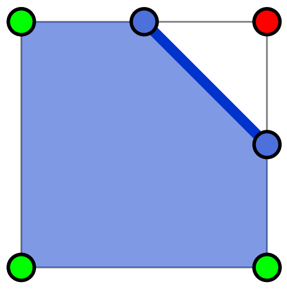

0110 = 6

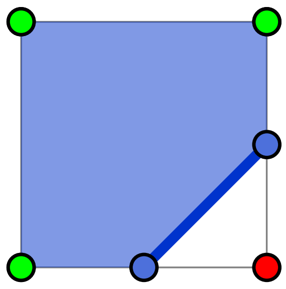

1100 = 12

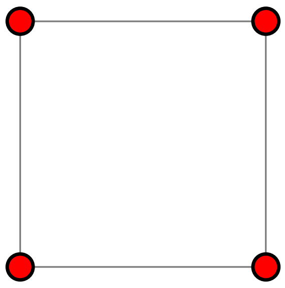

0011 = 3

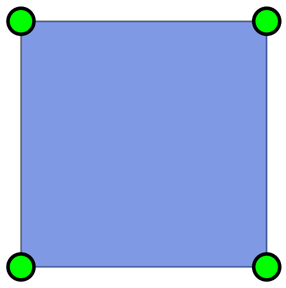

1110 = 14

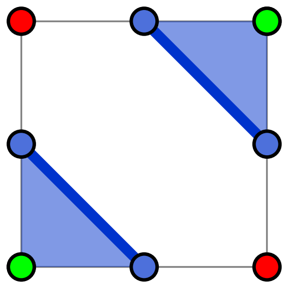

0111 = 7

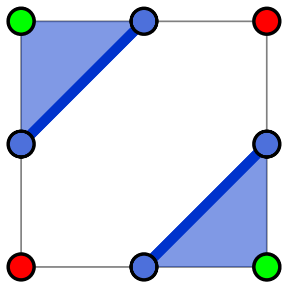

1011 = 11


1101 = 13


0000 = 0


1111 = 15


0101 = 5

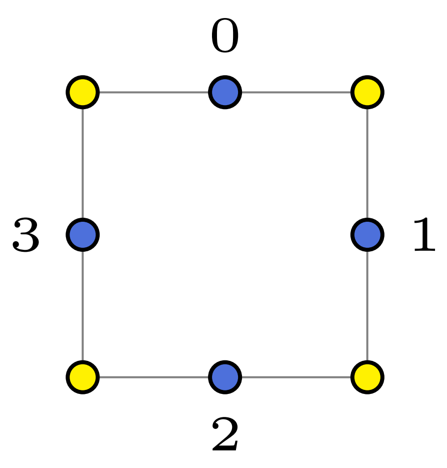

1010 = 10

We can also use bit-wise operations to construct a number encoding the the four Boolean values simultaneously. The key is to build up the number additively using **bitwise OR**.

```
int case = 0;
if (top-left corner is inside object) {
    case |= TOP_LEFT;
}
if (top-right corner is inside object) {
    case |= TOP_RIGHT;
}
if (bottom-right corner is inside object) {
    case |= BOTTOM_RIGHT;
}
if (bottom-left corner is inside object) {
    case |= BOTTOM_LEFT;
}
```

Now that we have a bitmap encoded as a number, we can use that number as an index into a **lookup table** to return the correct line segments. Of course, a lookup table is not unique and its ordering depends on the choice of encoding which corners as which bit index.

But how are we going to encode these line segments? Consider that out of all 16 possibilities, all line segments have end points which are midway between two corners. There are only 4 unique end points. Let us label them 0-3:



We will use vertex indices now to encode the line segments for each case. Since there at most two line segments among all the cases, let us encode each case as 4 indices. Where cases have only one line segment, pad the list with -1. Where cases have no line segments, make them a list of four -1.

```
int marching_squares_lut[16][4] = {
    {-1, -1, -1, -1},
    {2, 3, -1, -1},
    {1, 2, -1, -1},
    {1, 3, -1, -1},
    {0, 1, -1, -1},
    {0, 1, 2, 3},
    {0, 2, -1, -1},
    {0, 3, -1, -1},
    {0, 3, -1, -1},
    {0, 2, -1, -1},
    {0, 3, 1, 2},
    {0, 1, -1, -1},
    {1, 3, -1, -1},
    {1, 2, -1, -1},
    {2, 3, -1, -1},
    {-1, -1, -1, -1}
}
```

These indices can then itself be an index into a second lookup table for the vertices of the lines segments:

```
float g_verts[4][2] = {
    {0.5f, 1.0f},
    {1.0f, 0.5f},
    {0.5f, 0.0f},
    {0.0f, 0.5f}
};
```

Note these vertices do not give exact positions. Rather, they give the position of a line segment's vertex relative to the corners of the square which it is in. For a square with bottom-left corner (x, y) and a side length s, one can compute the absolute positions of a line segment's vertex as:

```
x + s * g_verts[i][0]
y + s * g_verts[i][1]
```

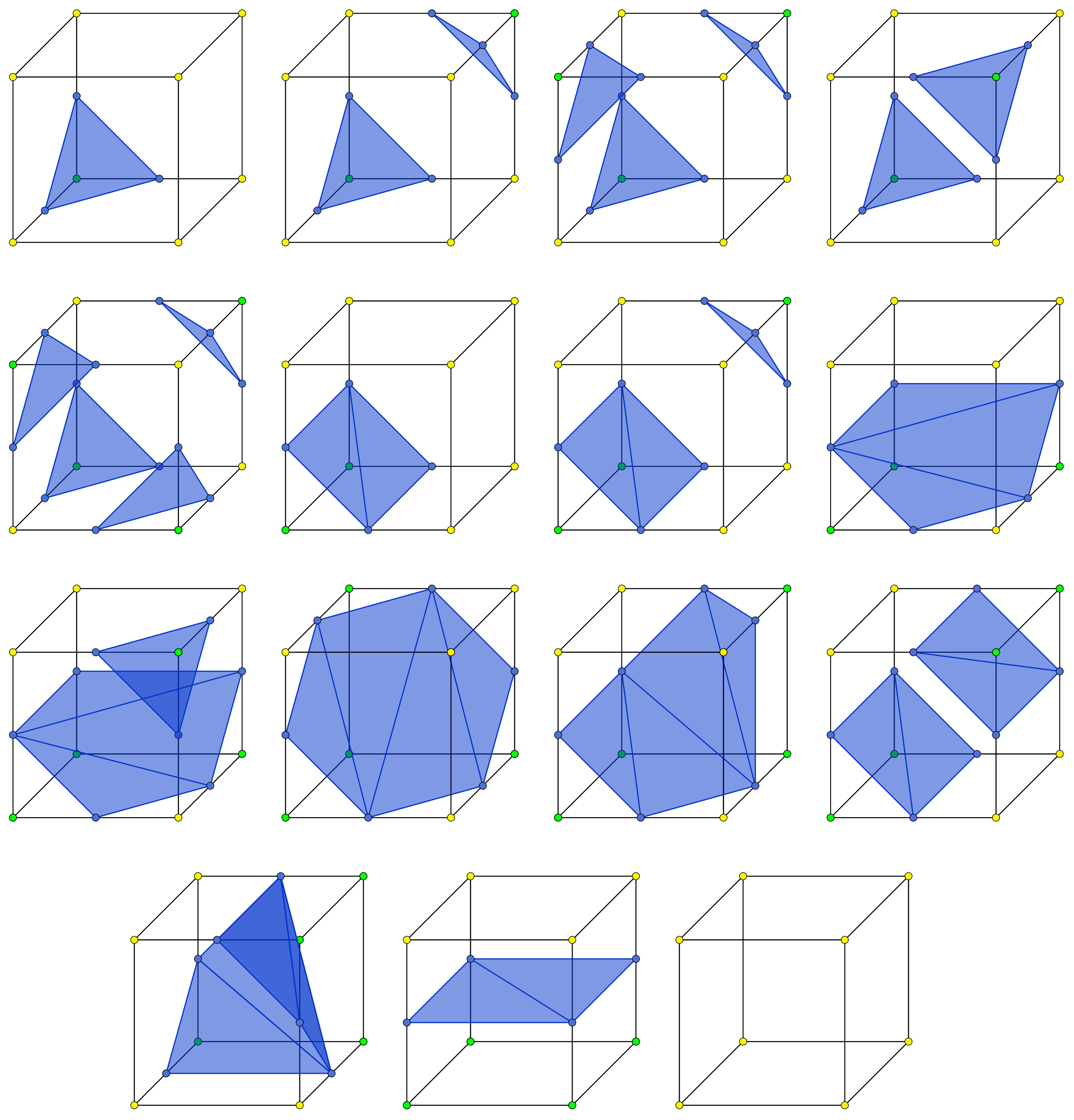
```
for (float y = ymin; y < ymax; y += s) {
    for (float x = xmin; x < xmax; x += s) {
        bottomleft = test(x,y);
        bottomright = test(x+s, y);
        topleft = test(x, y+s);
        topright = test(x+s, y+s);

        int case = 0;
        //build up bit mask...
    }
}
```

Put it all together: see MarchingSquares.cpp.

12.3 Marching Cubes

Now, we are going to extend all of this to the three dimensional case. As before, we are going to *march* through three-dimensional space, testing the corners of, now, a **cube**. For each corner that is inside or outside of the object, we fall into one of **256** cases. There are 8 corners of the cube and each is independently either inside of outside the object.

But, after excluding **rotational symmetries** and **inverse cases** we really only have 15 cases.



The marching cubes algorithm is not much different from the marching squares algorithm. We simply change the lookup table, and add an additional dimension to the “march”.

```
for (float z = zmin; z < zmax; z += s) {
    for (float y = ymin; y < ymax; y += s) {
        for (float x = xmin; x < xmax; x += s) {
            //test the 8 corners and build a bitfield
        }
    }
}
```