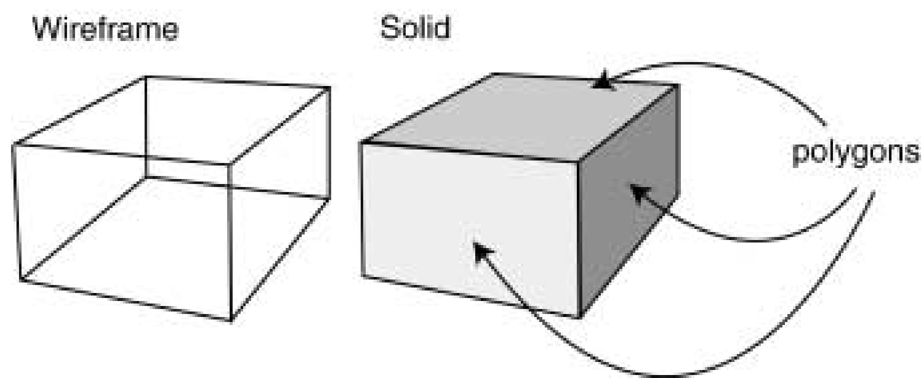
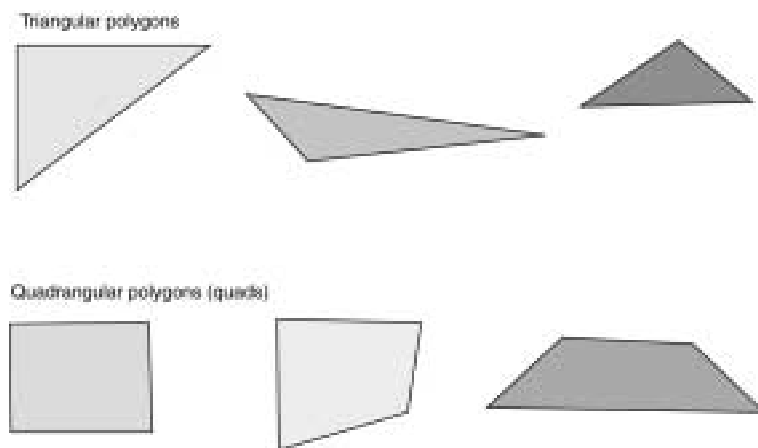
# Chapter 13
# Ultimate 3D

In recent years, as computers have become faster and more capable, it's become common to see images that obtain near total realism. In our wireframe examples, our 3D scenes were built up of several lines, and that's all. In typical 3D games like *Half-Life*, *Quake*, and *Tribes*, the scenes are built up of what is essentially a wireframe world, that's been made solid by stretching skins between the wireframe mesh.

Take a look at the following image to see what I mean:




These solid areas are known as polygons. A polygon is defined as an area that's enclosed within three or more lines. In the context of games, there are usually two types of polygon that are used to create 3D scenes – triangles and quadrangles (or quads, as they're usually called). Quads tend to be the polygon of choice in most games. Triangles are used more often in 3D movies and television when more detailed shapes are needed. A triangle allows you to have twice as much 3-dimensional detail as a quad:



Most of the 3D gaming world is made up of polygons, and indeed polygons have become the benchmark for measuring a game's performance; polygons per second rendered is a way of telling how powerful a 3D system is.

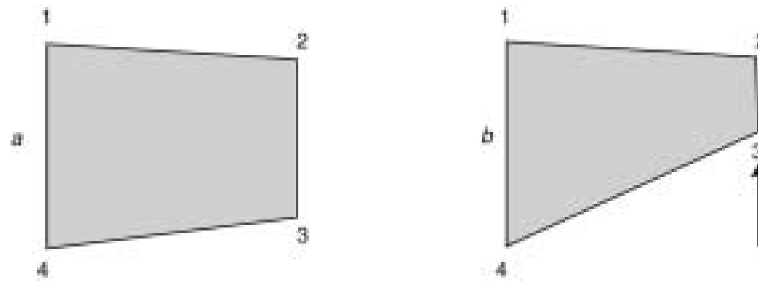
Now, with modern 3D games, most of the polygons are usually textured – the skin surface is a bitmap picture itself. The bitmap is squashed, rotated and scaled with perspective depending on the orientation of the polygon, to create the illusion that this object is an actual real-world 3D object, with surface texture and character:

In this texture mapped polygon, you can see that the surface bitmap has been rotated, skewed, scaled and stretched so that the top and side surfaces look like they're receding into the distance – it's been **mapped** onto the polygons. There are three polygons in this image, and in actuality, they're simply 2D screen objects. It's because of the way our eyes perceive them pieced together as a whole that we believe this object is completely 3D.

A game can have thousands of polygons on screen at once. A 3D CGI movie like *Toy Story (Disney-Pixar, 1995)*, *Final Fantasy (Square Pictures, 2001)*, or *Shrek (Dreamworks SKG, 2001)* can have hundreds of millions of polygons at once, but in Flash – how do we even create one polygon?

## Limitations of Flash

In the last chapter, I mentioned that Flash causes the annoying limitation of not being able to arbitrarily move vertices in a polygon. We saw how it was possible to move end points of a line (by moving the starting point, and stretching the line), but how do we take a solid shape with four vertices and move those vertices around to change the shape of our quad at runtime? Looking at this image, you can see the desired effect:




We want point 3 on the polygon to move, but we want points 1, 2, and 4 to remain in the same spot. How can we do this in an environment like Flash, where there are so few shape manipulation options available to us? Let's say that our polygon a is its own movie clip, how can we modify this movie clip to make it look like polygon b?

It's certainly possible to turn polygon a into our desired polygon b, at **design** time when you're building your FLA in Flash – all you have to do is drag that corner, simple, but what we're after is a way of converting polygon a into polygon b at **runtime**, using ActionScript. We're not talking about drawing a 4-sided **wireframe** square, we're talking about a **solid** shape made up of four lines and four vertices.

In pseudo-code we want to be able to draw our polygon on screen by providing four points to a polygon engine, like so:

```
draw_polygon (point1, point2, point3, point4, color);
```

In a programming language like C it's quite easy to do. And if you are using a programming library like DirectX, you can simply specify your points, specify your texture, and the perfect polygon will appear on screen exactly as you want it. However, in ActionScript – it can't be done that way.

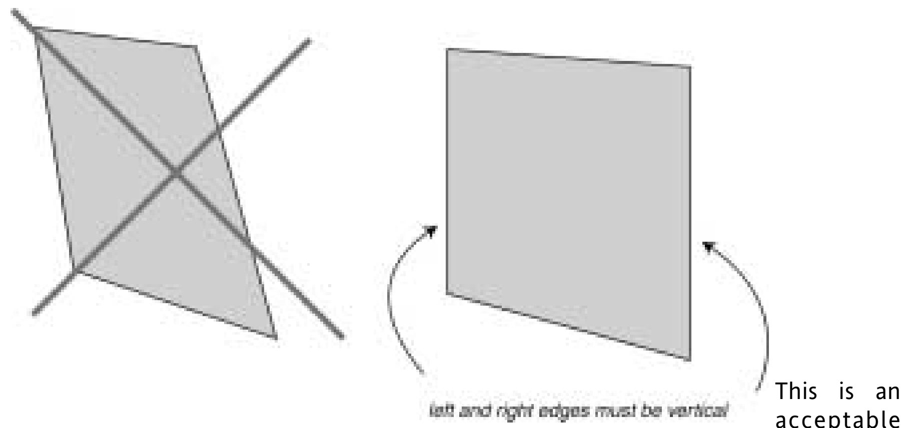
In fact, can it be done at all? Well, it's a long shot, but it might just work.

## The Flash Polygon Engine

We've seen how it's possible to create a great 3D wireframe game, but to a large extent, wireframe went out with the 80s. It's so retro! 21st Century gamers want **solid**. So, we have to come up with a solution in Flash for drawing arbitrary solid shapes on screen that don't need to be pre-rendered, rather they can be drawn as required at runtime, just like our wireframe images of the previous chapter.

This is where we have to be sneaky. After much head scratching, I have produced a basic polygon engine. However, since this is a considerably restricted and difficult thing for Flash to do, these are a few limitations that I must first mention:

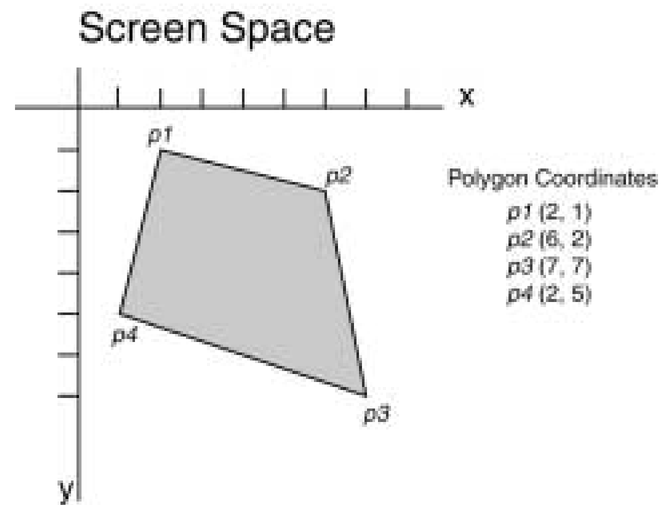
The side edges must be completely vertical.




This is an acceptable limitation however, because we plan on using this polygon engine for creating objects like walls that we can walk alongside.

The polygons cannot be texture mapped with detailed bitmaps. These polygons are going to be solid filled. Later, I'll show you how we can in fact make a few simple textures on the surfaces to give them a little bit of detail, but unfortunately we can't do detailed textures like modern commercial games. We just don't have the speed.

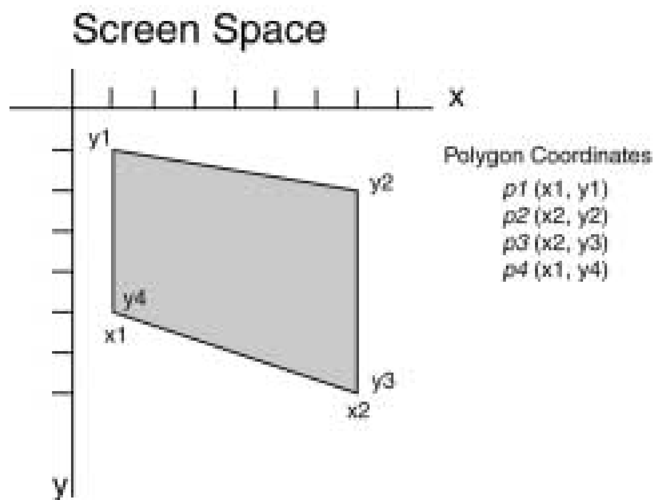
These things aside, we're still going to have a 3D world where solid polygons obscure other polygons in an arbitrary fashion. In a normal, unrestricted polygon, the information for its shape is contained as four distinct points. Let's look at a normal screen polygon:




This polygon could be created with the imaginary function:

```
draw_polygon (2, 1, 6, 2, 7, 7, 2, 5, RED);
```

Here, each point in the polygon is passed in order of x, y from point 1 to point 4, followed by the desired color. This would, of course, create a red quad. This function takes eight distinct numbers (x1, y1, x2, y2, x3, y3, x4, and y4), but as I said, Flash can't do this at runtime so we need to find a way of reducing the information it needs to take in. Let's take a look at our polygon, and how we could represent it:



As you can see I've chosen to use variables instead of numbers because I want to illustrate the fact that we have only *six* distinct variables. While we do have four y values (`y1, y2, y3, y4`), we only have two x values, left edge and right edge (`x1, x2`).

Now, for the purposes of making things clearer, I'm not going to refer to the y variables as simply `y1`, `y2`, `y3`, and `y4`. From now on I'm going to call them `y1t` and `y1b` for the top and bottom y values at `x1`, and `y2t` and `y2b` for the top and bottom values at `x2`.

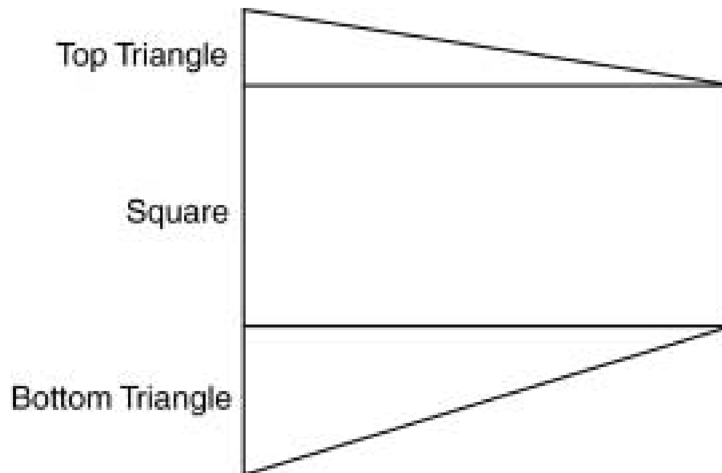
So, if we wanted to draw a polygon using this variable structure, we simply pass our x variables, and top and bottom y variables, like so:

```
draw_polygon (x1, y1t, y1b, x2, y2t, y2b, BLUE);
```

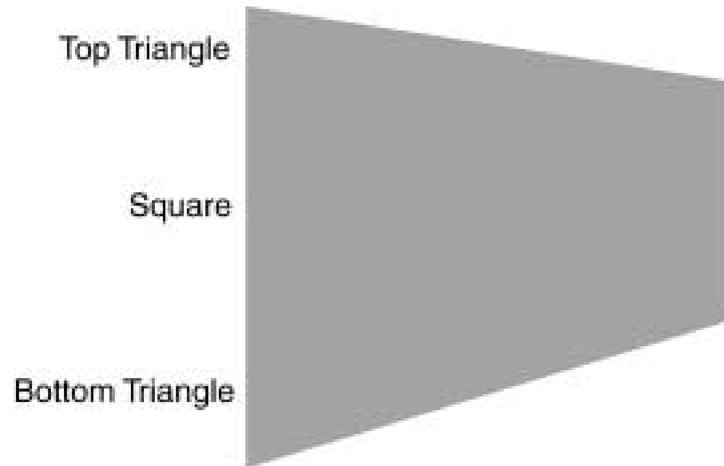
Using this, we could easily draw simple wall polygons, mapped from a 3D coordinate system. Now that you understand how the data is represented, let's take a look at how the graphics are actually drawn – the big secret.

## Abracadabra

Our answer comes in the form of finding an **arbitrary polygon**. Remember how we solved our wireframe problems by using a single line movie clip and then scaling it to create any line? Well, we're going to do the same thing here, but with a polygon instead of a line. We're going to base everything on the fact that a polygon with a vertical left and right edge is basically made up of three objects: one square and two triangles, like so:

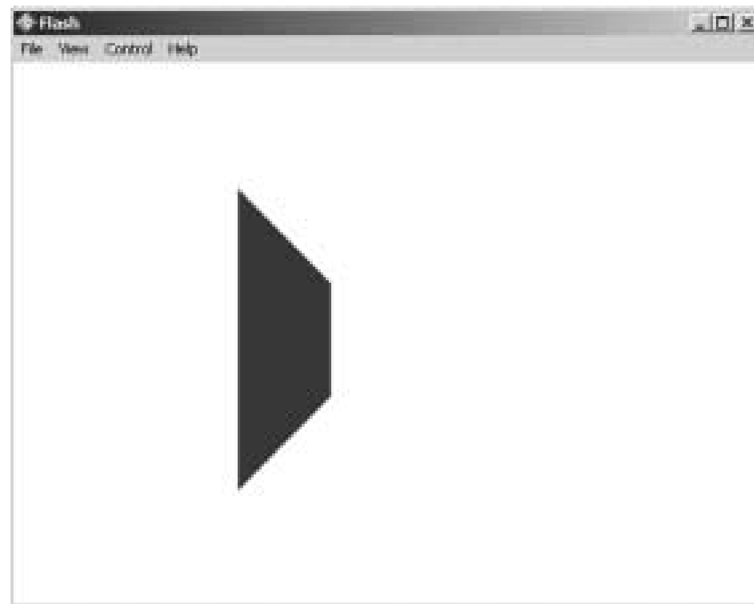


When the lines are removed, and the three shapes are filled in, we get this:

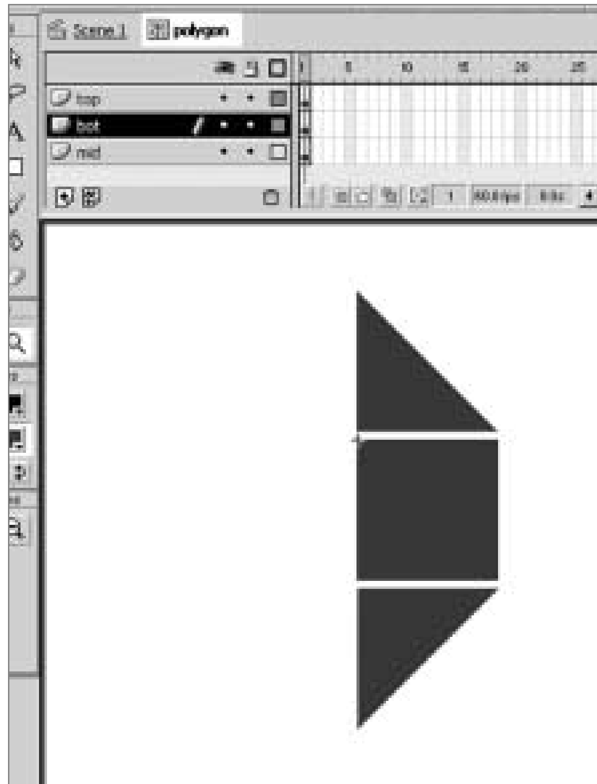



Our solid polygon! Remember that we can easily scale things in Flash using the `_xscale` and `_yscale` properties. What we're doing is scaling the two triangles and the square based on the information provided (`x1`, `y1t`, `y2b`, `x2`, `y2t`, `y2b`).

Let's take a look at the polygon engine running. Check out `demo13-1.fla`.



The key object in our movie is a movie clip on the main timeline called polygon, with the instance name poly0. It's this object that contains our three sub-objects, the top triangle, the square and the bottom triangle, each with a corresponding instance name of top, mid, and bottom:

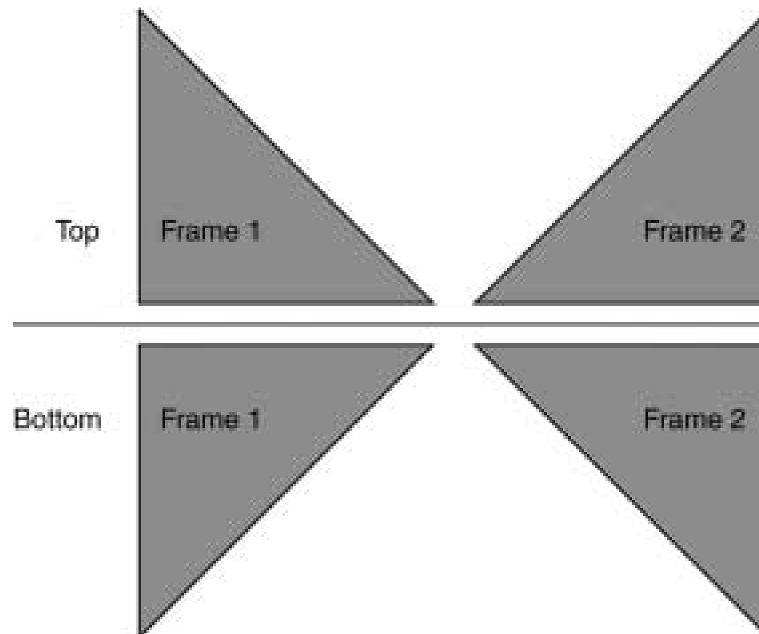



All three of the sub shapes are based on the size of 100x100. For example, the middle box is exactly 100x100 in size, while the top triangle is 100 along the base and the left side. It's almost like our line renderer from the previous chapter, except the triangle is filled in below the line.

We can use the _xscale and _yscale properties of mid to squash and stretch the middle rectangle easily to any width or height. Also, when we squash and stretch the top triangle, we can get many variations, like so:

All this is achieved by setting the `_xscale` and `_yscale` of `top`. Of course, the bottom triangle can also be squashed and stretched to produce similar versions of the top triangle, just upside down.

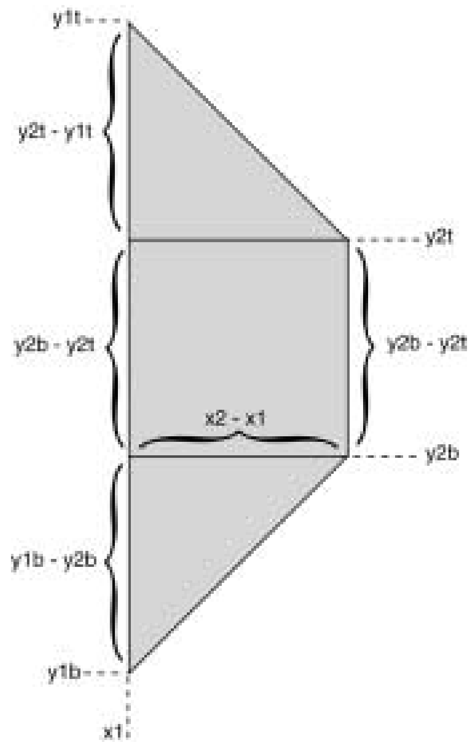
There is one more thing, while the `mid` movie clip is only one frame, the `top` and `bottom` movie clips are both two frames. Frame one contains the triangle going from left to right, and frame two contains the triangle going from right to left, like so:



So, now you can see exactly how our polygon is defined and laid out, let's take a look at the actual code required to make it appear anywhere on screen. Let's assume that the instance name of our polygon movie clip is contained within a variable called obj:

```
eval(obj).mid._x = x1;
eval(obj).mid._xscale = x2 - x1;
eval(obj).mid._y = y2t;
eval(obj).mid._yscale = y2b - y2t;

eval(obj).top._x = x1;
eval(obj).top._y = y1t;
eval(obj).top._xscale = x2 - x1;
eval(obj).top._yscale = y2t - y1t;

eval(obj).bottom._x = x1;
eval(obj).bottom._y = y2b;
eval(obj).bottom._xscale = x2 - x1;
eval(obj).bottom._yscale = y1b - y2b;
```

Remember, x1, y1t, y1b, x2, y2t, and y2b are our six variables as defined earlier. That's all we need to make the polygon appear on screen correctly:



We do have to take a few other things into consideration. If it happens that y2t is more than y1t (the top slopes up and to the right, instead of down to the left), then we must switch `top` to frame 2. Correspondingly, if y1b is less than y2b, then we must switch `bottom` to frame 2. This is handled with the following code:

```
if (y1t <= y2t)
{
        eval(obj).top.gotoAndStop(1);
        ry1t = y1t;
        ry2t = y2t;
}
if (y1t > y2t)
{
        eval(obj).top.gotoAndStop(2);
        ry1t = y2t;
        ry2t = y1t;
}
if (y1b > y2b)
{
        eval(obj).bottom.gotoAndStop(1);
        ry1b = y1b;
        ry2b = y2b;
}
if (y1b <= y2b)
{
        eval(obj).bottom.gotoAndStop(2);
        ry1b = y2b;
        ry2b = y1b;
}
```

This means that we can have polygons of any shape, as long as the left and right edges are both perfectly vertical. Finally, we must make sure that the center point of `poly0` is sitting at (0,0) on the main timeline. This is because we're working on a local coordinate system to move `top`, `mid` and `bottom` around, and when we move them, they are being moved relative to the location of `poly0`.

Looking at the previous code, you can see that we're reassigning the six passed screen variables `x1`, `y1t`, `y1b`, `x2`, `y2t`, and `y2b` into six new variables `rx1`, `ry1t`, `ry1b`, `rx2`, `ry2t`, and `ry2b` if we need to play frame 2 of either triangle. For example, if `y1t` is more than `y2t` then we must set `ry1t` to `y2t` and `ry2t` to `y1t` so that all of our math will work correctly.

The whole polygon rendering routine looks like this:

```
function renderpoly(x1, y1t, y1b, x2, y2t, y2b, avez, obj)
{
     // Takes a poly movie clip and makes it appear on screen
     // this is the magic routine.  It takes 6 screen coordinates
     // Left edge, Right edge, and top and bottom at both edges

     // Figure out which top point (near or far) is the highest
     if (y1t <= y2t)
     {
            _root[obj].top.gotoAndStop(1);
            ry1t = y1t;
            ry2t = y2t;
     }
     if (y1t > y2t)
     {
            // flip to frame 2, the triangle facing left
            _root[obj].top.gotoAndStop(2);
            ry1t = y2t;
            ry2t = y1t;
     }

     // Figure out which bottom point (near or far) is the lowest
     if (y1b > y2b)
     {
            _root[obj].bottom.gotoAndStop(1);
            ry1b = y1b;
            ry2b = y2b;
     }
     if (y1b <= y2b)
     {
            // flip to frame 2, the triangle facing left
            _root[obj].bottom.gotoAndStop(2);
     ry1b = y2b;
            ry2b = y1b;
     }

     rx1 = x1;
     rx2 = x2;

     // adjusted values now in rx1, rx2, ry1t, ry1b, ry2t, ry2b

     // Scale middle square of the polygon based on screen values

     _root[obj].mid._x = rx1;
     _root[obj].mid._xscale = rx2 - rx1;
     _root[obj].mid._y = (ry2t - 1);
     _root[obj].mid._yscale = (ry2b - ry2t) + 2;
```

```
        // scale the top triangle based on the screen values
        _root[obj].top._x = rx1;
        _root[obj].top._y = ry1t;
        _root[obj].top._xscale = rx2 - rx1;
        _root[obj].top._yscale = ry2t - ry1t;

        // scale the bottom triangle based on the screen values
        _root[obj].bottom._x = rx1;
        _root[obj].bottom._y = ry2b;
        _root[obj].bottom._xscale = rx2 - rx1;
        _root[obj].bottom._yscale = ry1b - ry2b;

        // Set the depth level of the polygon based on avez,
        // The average z depth of the poly (passed in from renderer)
        _root[obj].swapDepths(Math.round(10000 - avez));

    }
```

That's the magic routine that makes our solid polygons happen in Flash. You'll notice that we're passing in `obj`, which is just the name of the movie clip instance containing our polygon.

Also, since this polygon is actually going to be in 3D space, we're passing in another variable called `avez`. This is the "average z" of the polygon. You'll see how we get this later, but it's basically the z value that's calculated when the z value at both ends of the polygon are averaged. This is used to order our polygons on screen so that they are rendered the correct distance from the viewer.

## Making it 3D

If you look at `demo13-1.swf` you'll see a single polygon on screen, moving around in 3D. How do you make the polygon into something that's 3D? Well, to be perfectly honest, you don't. The polygon render engine knows nothing about 3D (except for the `avez`, but that's only for layering). The polygon engine is brought into action only *after* any 3D walls/quads/polygons have been translated, rotated and then projected into screen space. Our polygon engine only creates 2D representations of 3D objects, just like our wireframe engine from the previous chapter.

Looking at the demo, you'll see a movie clip on the main timeline called controller. Attached to this movie clip, on the `load ClipEvent`, you'll see the following code:

```
// View position
xloc = 0;
yloc = 0;
zloc = -10;
d = 200;

// Polygon information
x1 = 0;
y1t = -100;
y1b = 100;
z1 = 100;

x2 = 0;
y2t = -100;
y2b = 100;
z2 = 400;
```

For this demo, we're defining one 3D polygon. You can see that we've added variables called `z1` and `z2`. This is how we're giving our polygon its third dimension. Attached to the controller movie clip's `enterFrame ClipEvent`, you can find the following code:

```
onClipEvent(enterFrame)
{

     nm = "poly0";

     // Translate polygon into world space, relative to
     // view (xloc, yloc, zloc)
     wx1 = x1 - xloc;
     wy1t = y1t - yloc;
     wy1b = y1b - yloc;
     wz1 = z1 - zloc;

     wx2 = x2 - xloc;
     wy2t = y2t - yloc;
     wy2b = y2b - yloc;
     wz2 = z2 - zloc;

     // Project all 3D points onto the screen
     // to determine our 2D screen points - move to
     // screen center (275, 200)
     sx1 = (d * wx1 / wz1) + 275;
     sy1t = (d * wy1t / wz1) + 200;
     sy1b = (d * wy1b / wz1) + 200;

     sx2 = (d * wx2 / wz2) + 275;
```

*continues overleaf*

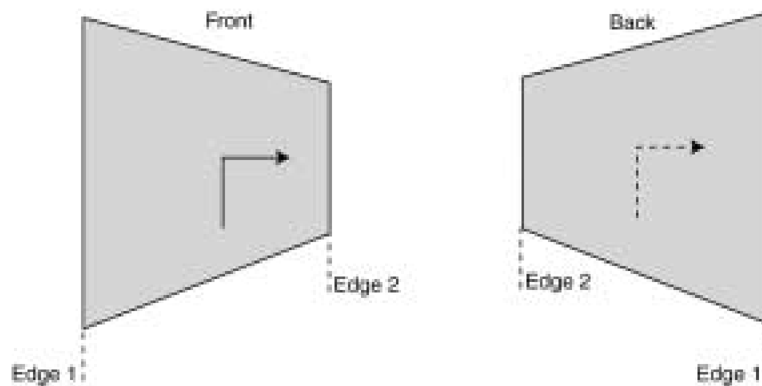
```
            sy2t = (d * wy2t / wz2) + 200;
            sy2b = (d * wy2b / wz2) + 200;

            // If the left edge is greater than our
            // right edge, then the polygon is backwards
            // so we must swap left values and right values
            if (sx1 > sx2)
            {
                   tsx1 = sx1;
                   tsy1t = sy1t;
                   tsy1b = sy1b;

                   sx1 = sx2;
                   sy1t = sy2t;
                   sy1b = sy2b;

                   sx2 = tsx1;
                   sy2t = tsy1t;
                   sy2b = tsy1b;
            }

            // Render the 2D polygon we've come up with
            renderpoly(sx1, sy1t, sy1b, sx2, sy2t, sy2b, 0, nm);

            // Move the camera around in smooth circles
            xloc+= 3 * Math.cos(ang+=.005);
            zloc-= Math.sin(ang+=.005);
       }
```

We're simply taking our previously defined 3D polygon, translating it by `xloc`, `yloc`, and `zloc`, and then perspective-projecting it on screen. By dividing all the x and y coordinates by the corresponding z coordinate, we will create the 3D we're trying to achieve. This is the same principle we covered in the last chapter with projecting 3D points into 2D to render our wireframe images.

Remember, now that we're in 3D, the terms "left" edge and "right" edge, are not as applicable, because edges can also be "near" and "far" and be on the same x coordinate. For this reason, I will refer to them as edge 1 (x1, y1t, y1b, z1) and edge 2 (x2, y2t, y2b, z2).

We've still got one more thing to do though; we want to check whether or not we're looking at the back of our polygon. The following code ought to do the trick, by checking to make sure that edge 1 hasn't somehow actually moved right of edge 2 on screen:

```
if (sx1 > sx2)
{
       tsx1 = sx1;
       tsy1t = sy1t;
       tsy1b = sy1b;

       sx1 = sx2;
       sy1t = sy2t;
       sy1b = sy2b;

       sx2 = tsx1;
       sy2t = tsy1t;
       sy2b = tsy1b;
}
```




This check is useful in determining the visibility of polygons when we're rendering solid objects. Take a cube for example – any of the surfaces facing away from us (backfaces) will have their edge 1 to the right of their edge 2 on screen. This little piece of logic means that the edge is facing away from us, and we may not want to draw it.

In the above piece of code however, what we're actually doing is checking to see if edge 1 has surpassed edge 2, and if it has, we're swapping all the `sx` and `sy` values of both edges, flipping the polygon to face us, without changing the way it looks.

We could change the logic like this:

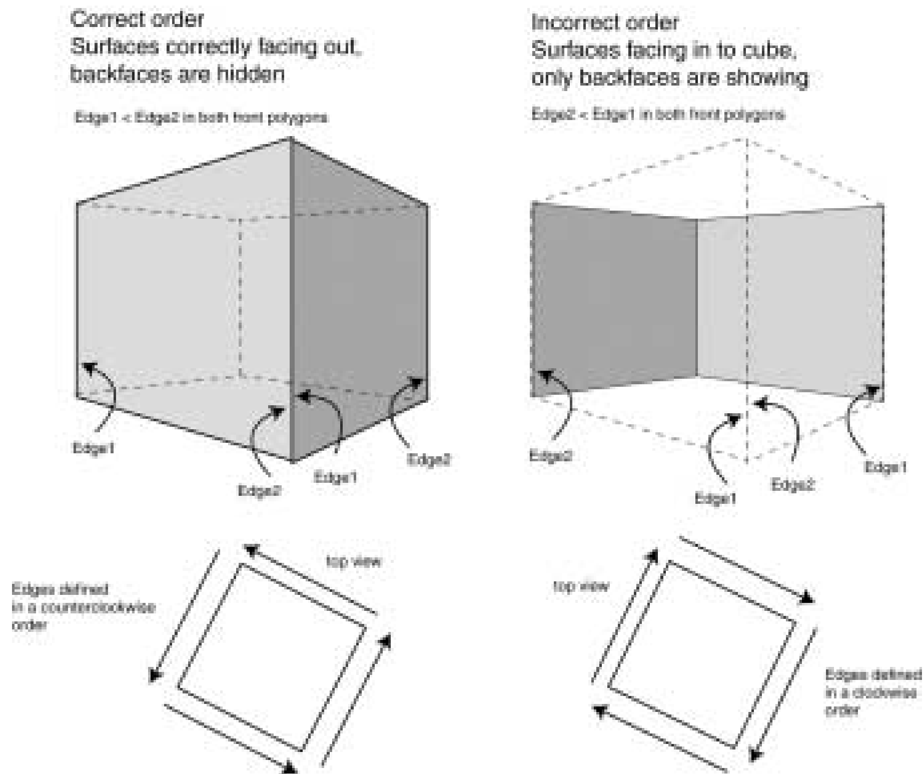
```
offscreen = false;

if (sx1 > sx2)
{
       if (culling)
              offscreen = true;
       else
       {
       tsx1 = sx1;
       tsy1t = sy1t;
       tsy1b = sy1b;

       sx1 = sx2;
       sy1t = sy2t;
       sy1b = sy2b;

       sx2 = tsx1;
       sy2t = tsy1t;
       sy2b = tsy1b;
}
}
```

This way, we can create a flag called `culling` that's `true` or `false`. If it's set to `true`, then we have a variable called `offscreen` that we can use later to determine whether or not to actually draw our polygon, or hide it, like so:

```
if (offscreen)
{
       _root[nm]._visible = false;
}
else
{
       // Draw, and call renderpoly with our 6 screen
       // coordinates
       _root[nm]._visible = true;
       renderpoly(sx1, sy1t, sy1b, sx2, sy2t, sy2b, avez, nm);
}
```

You'll see all of this later in our 3D game demonstration, but I'm giving you a teaser for it now. **Backface culling** is a term used in professional 3D game and movie production. It's a form of hidden surface removal, where we're trying to cut down the amount of drawing that a computer must do, in order to make our game run faster. If you look at `demo13-2.swf`, you'll see the culling in action. It's exactly like `demo13-1`, except when the back of the polygon is facing us it will disappear.

It's important to make note of just which edge you define as edge 1 and edge 2. If we define out polygons in the wrong order, we could make a cube where all the edges appear to face in, instead of out, and we'd see something like this:




In the right hand image, the object will look quite strange as we walk around it and only the rear polygons are visible! Essentially, we want to make sure that our polygons are defined in an order that goes *around* the object – either clockwise or counterclockwise. Take a look at this polygon definition:

```
x1 = 0;
y1t = -100;
y1b = 100;
z1 = 100;

x2 = 200;
y2t = -100;
y2b = 100;
z2 = 100;
```

This is a flat polygon that's facing us, the y and z values of both edges are identical, however, x1 is at 0, and x2 is at 100. Without moving, we'll see this polygon facing us because x1 is less than x2. However, something rather different happens if we change the definitions to look like this:

```
x1 = 200;
y1t = -100;
y1b = 100;
z1 = 100;

x2 = 0;
y2t = -100;
y2b = 100;
z2 = 100;
```

Mathematically speaking, that polygon is pretty much the same as the previous, except for one key difference – x2 is less than x1 (edge 2 is less than edge 1), so the polygon is facing away from us, and we therefore won't draw it.

Because our culling check starts with this:

```
if (sx1 > sx2)
```

...we want to define our shapes in a counterclockwise direction, as in the left box in the previous image. However, if our check looked like this:

```
if (sx1 < sx2)
```

...then we would only see backfaces if they were defined in a counterclockwise direction. Therefore, using this check, we must define our objects in a clockwise direction.

It's important to remember that we're looking at polygon orientation in screen space, and not simply in world space. We only do the backface check once we've transformed and projected the polygon into screen coordinates.

## Speed Considerations for Games

There are some important things to note as we delve deeper into this type of 3D for games. We're taxing Flash just about as much as we possibly can. We're going to have graphics that are updating full screen, every frame. The more polygons we have on screen (even if they're partially hidden) the more our game's performance is going to suffer. So, we need to take a look at how we can reduce what we're demanding of Flash.

## Limiting the Math

In the previous chapter, we showed you how to rotate the object, and rotate the world through a series of detailed and complex equations. The problem with that is that in the context of a game, we're going to find that much math slows down performance considerably. So what can we do?

Well, for a start we can make sure that our game only does what math is necessary. Let's assume that our game is made up of a series of stationary boxes. In our previous chapter, we had a function called manipulateobject, which took a 3D object and rotated it around three axes and translated it into position, then we used a function called drawobject, which rotated the object around the player's point of view:

```
manipulateobject(cube, offx, offy, offz, playerrx, playerry, playerrz);
drawobject(cube);
```

To optimize our game, we're going to eliminate the rotation in the `manipulateobject` step, and combine the translation relative to the player into the draw routine. This way, we only need to perform one rotation (relative to the player) and two translations (the object's position plus our position). Translations are easy because they're simply additions, but the elimination of the rotation steps is crucial. We're also going to limit the amount of rotation that the player can do to only one axis, only letting them perform a yaw around the y-axis – turning left and right.

To recap:

- We're assuming that the object won't need to rotate around its own axis – meaning that the box won't be spinning on the spot. This means we assume its coordinates are provided as-is, and do not need to be moved. ***We're going to store the object in variables in its already manipulated state***.
- We're assuming that all translation will simply be relative to the player.
- We're assuming that the player will only turn around one axis (the y-axis) thus reducing the amount of math required to compute rotations.

So, now we have a grip on that, let's see if we can't just impress ourselves and put it all into a game!

Introduction
chapter 1
chapter 2
chapter 3
chapter 4
chapter 5
chapter 6
Case Study 1
chapter 7
chapter 8
chapter 9
chapter 10
chapter 11
chapter 12
chapter 13
Case Study 2
chapter 14
chapter 15
Director Afterword