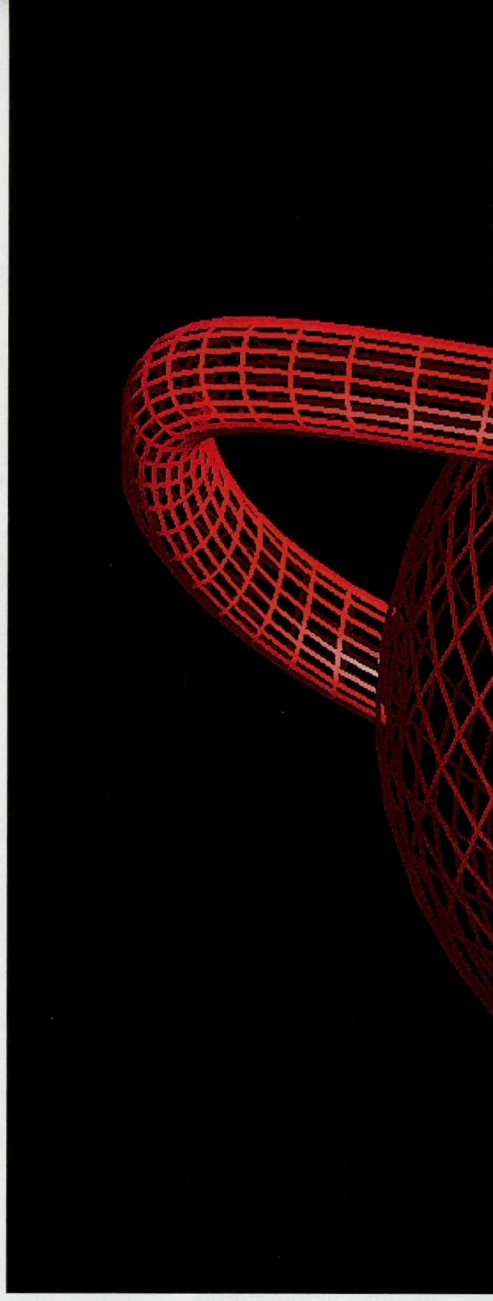
DOI:10.1145/2428556.2428571

Article development led by acmqueue
queue.acm.org

It is easy to do amazing things, such as rendering the classic teapot in HTML and CSS.

BY BRIAN BECKMAN AND ERIK MEIJER

The Story of the Teapot in DHTML



BEFORE THERE WAS Scalable Vector Graphics (SVG), Web Graphics Library (WebGL), Canvas, or much of anything for graphics in the browser, it was possible to do quite a lot more than was initially obvious. To demonstrate, we created a JavaScript program that renders polygonal 3D graphics using nothing more than HTML and CSS. Our proof-of-concept is fast enough to support physics-based small-game content, but we started with the iconic 3D "Utah teapot" (Figure 1) as it tells the whole story in one picture. (For background, see http://bit.ly/KQK9a.) It is feasible to render this classic object using just regular `<div>` elements, CSS styles, and a bit of JavaScript code (Figure 2). This tiny graphics pipeline serves as a timeless demonstration of doing a lot with very little.

The inspiration for this project came from Web developer Jeff Lau, who on his blog UselessPickles implemented a textbook graphics pipeline in handwritten JavaScript (http://www.uselesspickles.com/triangles/) Lau's demo embodies a glorious hack for efficiently rendering triangles in HTML, shown in Figure 3.

The glorious hack is explained later, but, to spoil the punch line, once you have arbitrary triangles, you can easily render arbitrary polygons, and thus arbitrary polygon-based models. The only remaining issues for game-competent 3D graphics are texture mapping, bump mapping, reflection mapping, and performance. These various kinds of mappings all require pixel-based primitives: the ability to render individual pixels efficiently. Though it is *possible* to render individual pixels using just the `<div>` element with a CSS style to shrink the element to pixel size, this obviously does not provide sufficient performance for classic scan conversion of 3D models. The work to render individual pixels is quadratic in

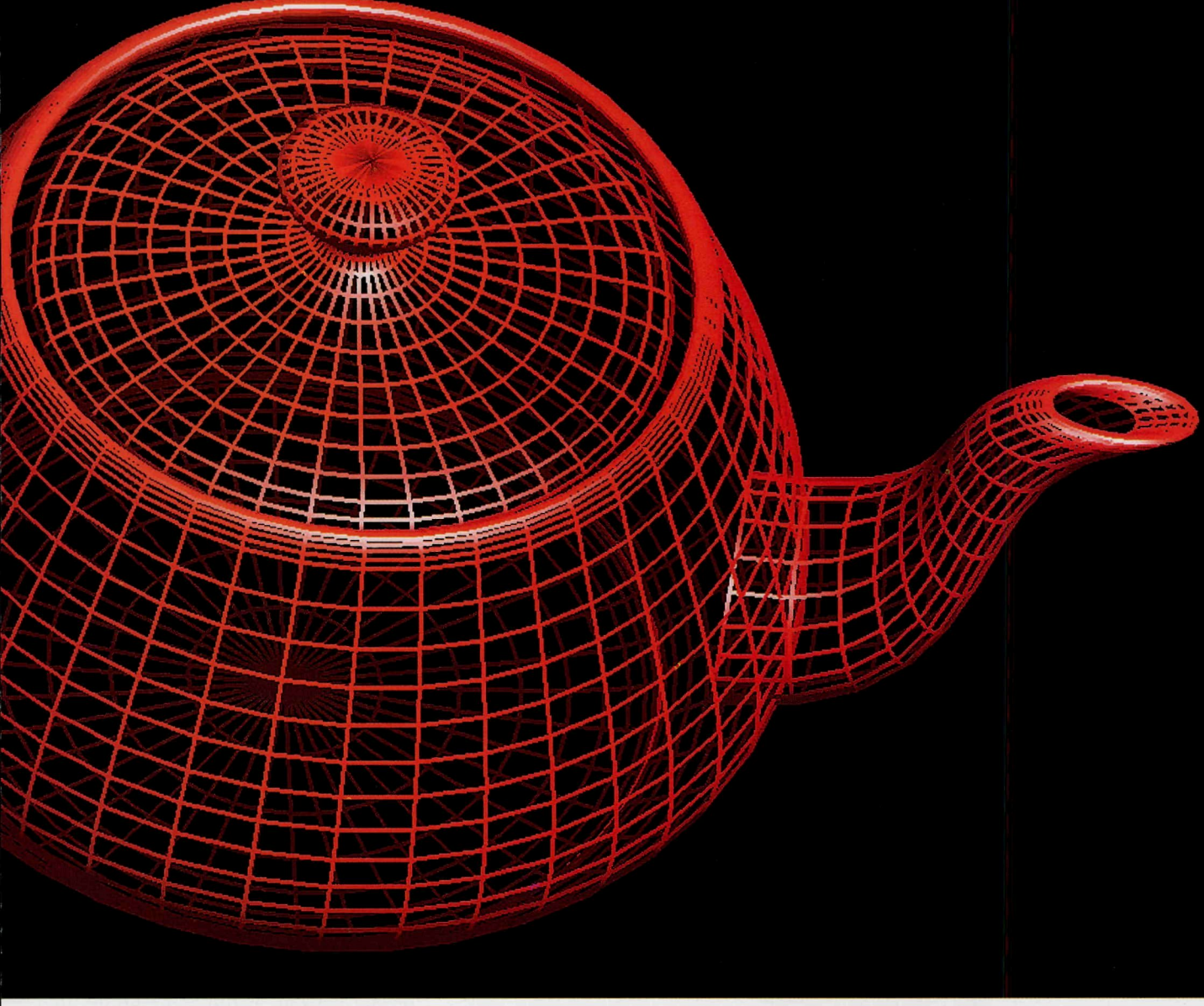


IMAGE BY LINGFA YANG

the linear size of a 2D figure, meaning that doubling the size of a figure requires roughly four times the work.

The elements, however, also begrudgingly provide a way to draw vertical and horizontal lines, if, for no other reason, than for borders around text. Several other bloggers (including David Betz and the late Walter Zorn) noted that by decomposing a figure into parallel "raster lines" instead of pixels, work is linear in the linear size of the figure, meaning that doubling the size only doubles the work. They created JavaScript 2D graphics libraries with reasonable performance by combining *pixel drawing where necessary and line drawing where possible* in s.

The following is an HTML page that illustrates the linear method by drawing a right triangle of eight vertical raster lines of linearly increasing height:

```
<style>div{ background:Black;
position:absolute; width:9px; }
</style>
<div style="left:10px;
height:10px;"></div>
<div style="left:20px;
height:20px;"></div>
<div style="left:30px;
height:30px;"></div>
...
<div style="left:80px;
height:80px;"></div>
```

The CSS style sheet and the inline position and height declarations create eight instances of with linearly increasing left coordinate and linearly increasing height. This HTML page renders as shown in Figure 4.

The linear pattern of coordinate and height values in HTML should be obvious. It should also be obvious how to write a program to generate a similar HTML page that renders not only right triangles with a scheme like this: just arrange the coordinate and dimension values to be linearly increasing or decreasing in appropriate ways. The following program dynamically generates elements exactly as the static markup:

```
<script>
    for(var i = 1; i < 9;
    i++)
      with(document)
      with(body.appendChild(
      createElement("div")).
      style)
    {
       left = i * 10;
       height = i * 10;
    }
</script>
```

Figure 1. The "Utah Teapot" (from "Fast Ray Tracing by Ray Classification," by James Arvo and David Kirk, 1987).

Figure 2. The teapot rendered in HTML, CSS, and JavaScript.

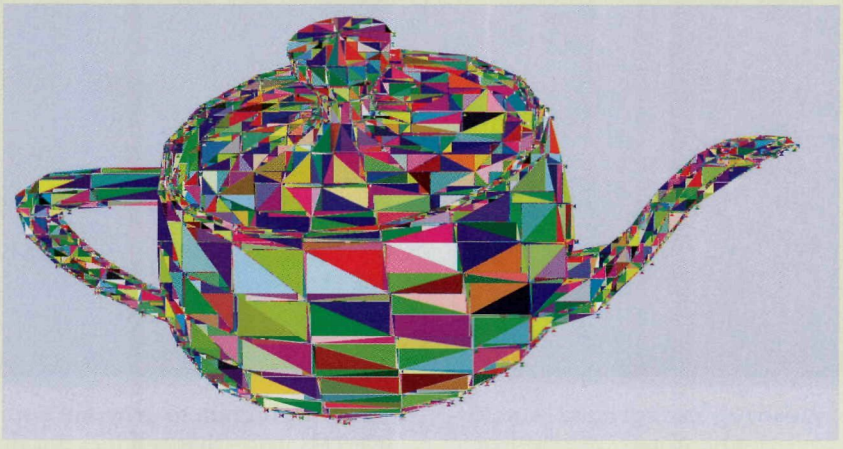
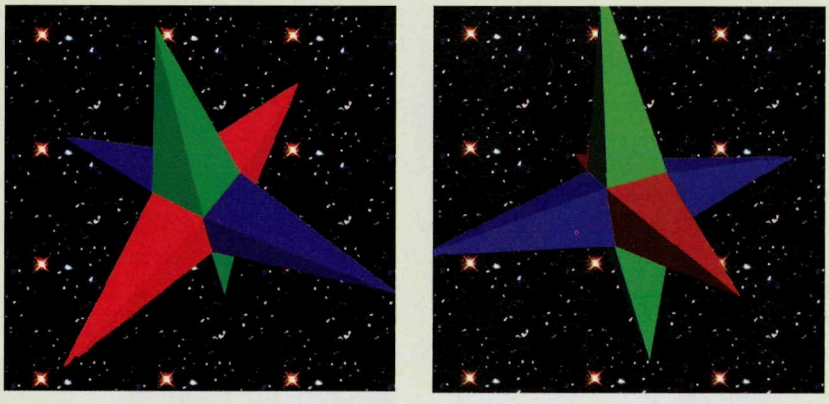


Figure 3. Lau's DHTML demo.



Standard rasterizer algorithms such as Bresenham's permit drawing all kinds of figures. Any programmer who has taken Computer Graphics 101 has seen enough now to create an entire workable 2D graphics library in HTML.

Logarithmic Performance

It is possible to achieve quadratic performance from pixels and linear performance from rasters. Can we do better than linear? Lau found *logarithmic* performance, meaning that doubling the linear size of a figure requires only a constant amount of more work, usually just one or two more calls to primitives. Logarithmic is *much* more efficient than linear. The difference is the same as that between binary search and linear search. Lau noticed that `<div>` + CSS has a subtly hidden primitive right triangle, if you know where to look. Then he presented a beautiful way to decompose an arbitrary triangle into a logarithmic number of right triangles: his glorious hack.

Notice in Figure 5 that HTML allows rendering the four borders of a `<div>` completely independently by setting the `border-XXX` colors. Setting the width of the `<div>` to zero removes the text, leaving just four triangles, as in Figure 6.

Is it possible to get rid of two of the triangles? This is straightforward: make the width of the right (yellow) border zero as in the "animation" in Figure 7, and similarly shrink the bottom (blue) border to make it disappear as well, as in Figure 8; this leaves just a green and a red right triangle.

Figure 4. Render of linear method.

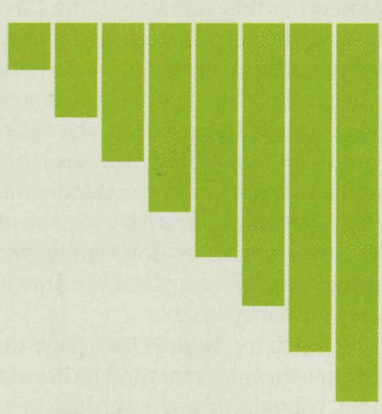


Figure 5. HTML border colors.

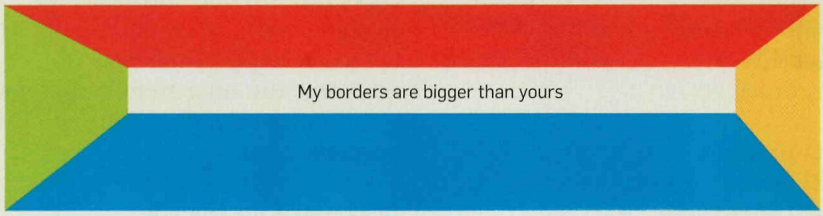


Figure 6. Triangles in HTML.

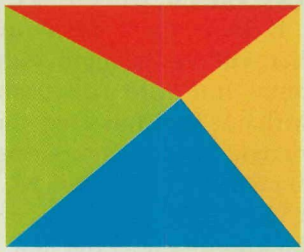


Figure 7. Shrinking the first triangle.

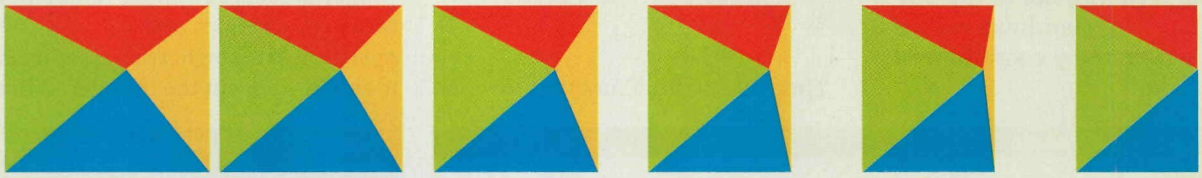
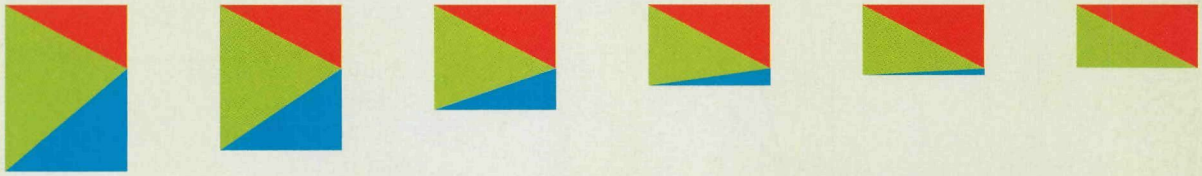


Figure 8. Shrinking the second triangle.



Now, setting one of the remaining border colors to "transparent" renders a single right triangle at the native efficiency of the underlying browser's rendering engine, presumably very high (Figure 9).

Setting the left and bottom borders' width to zero and making the other appropriate borders transparent—straightforward extensions of Lau's method—produces HTML primitives for all four kinds of right triangles, as in Figure 10.

Assume at this point a JavaScript function `drawRightTriangle(P1, P2, P3)` that can render any of these right triangles given the (coordinates of the) three vertices. The details are tedious and unenlightening, but suffice it to say that each of the four branches in the implementation must set the proper attributes of an underlying `<div>` tag, either directly or through CSS style classes.

Now we have really fast *right* triangles, but where are the *arbitrary* triangles with logarithmic performance, where doubling the triangle size means just one or two extra calls to the right-triangle primitive? Lau's original code is iterative in style, but the underlying recursive description is elegant, as follows:

Consider an arbitrary triangle; by definition of a triangle, the three vertices are *not* all on the same line. There are just two cases to consider: either

Figure 9. Making the third triangle transparent.

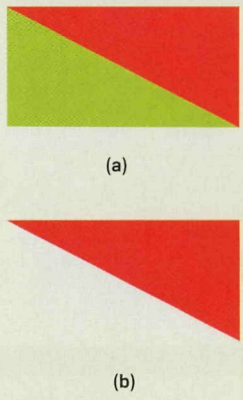
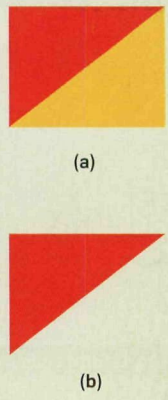


Figure 10. HTML primitives for all four kinds of right triangles.



there is one horizontal leg, or there is not (Figure 11). If there is one horizontal leg, then skip to the next paragraph. If there is not one horizontal leg, then cut the triangle with one horizontal line into *two* triangles, each with one horizontal leg. Cutting the triangle means computing the coordinates of a new point, `P4`, as in Figure 12.

The `y` coordinate of the new point `P4` is the same as the `y` coordinate of the middle-in-`y` point, `P2`—that is, `P4.y == P2.y`—and the `x` coordinate of the new point is proportionately as far from the `x` coordinate of the bottom point as the `y` coordinate of the new point is from the `y` coordinate of the bottom point:

```
P4.x == P3.x+(P1.x-P3.x)((P4.y-P3.y)/
(P1.y-P3.y))
```

The pseudocode, assuming that `P1`, `P2`, and `P3` are in downward, increasing-`y` order, is:

```
function
drawTriangleWithoutHorizont
alLeg(P1, P2, P3)
{  ... compute P4 according
   to equations above ... ;
   drawTriangleWithOneHori
   zontalLeg(P1, P2, P4);
   drawTriangleWithOneHori
   zontalLeg(P3, P2, P4);
}
```

There is one final function to write: `drawTriangleWithOneHorizontalLeg`. Recall that the two kinds of triangles-with-one-horizontal-leg are hanging-down and standing-up. They are completely symmetric, so let us work out the final steps only for the standing-up triangle. There are three possible cases:

- The tip is between the two base vertices—an *acute* triangle.
- The tip is exactly over one of the two base vertices—a *right* triangle.
- The tip is either to the right or the left of the base segment—an *obtuse* triangle.

If *acute*, cut the triangle vertically into two right triangles and call it a day! If *right*, well, it is a right triangle and done! If *obtuse*, then cut the triangle vertically

Figure 11. Two types of triangle.

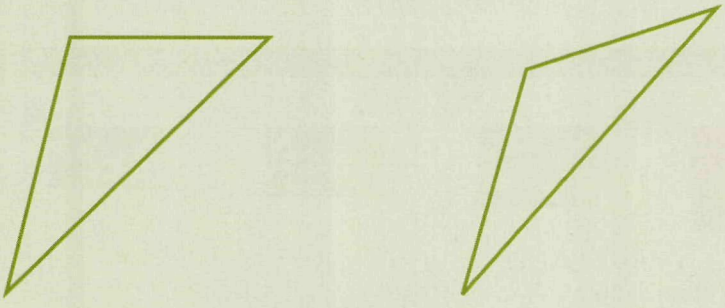


Figure 12. Forcing a horizontal side of the triangle.

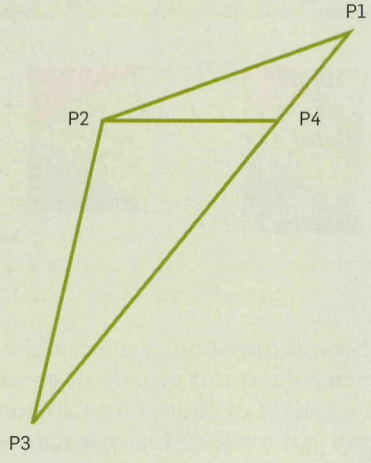


Figure 13. Forcing a vertical side of the triangle.

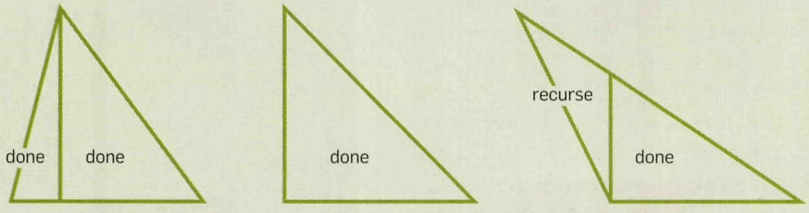


Figure 14. The decomposition of a triangle.

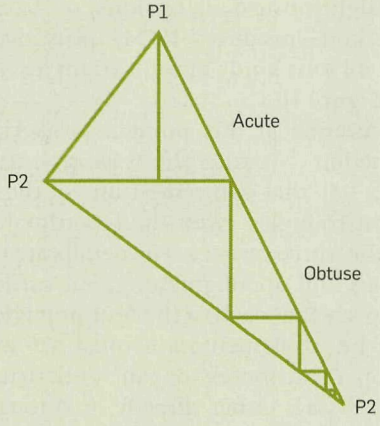
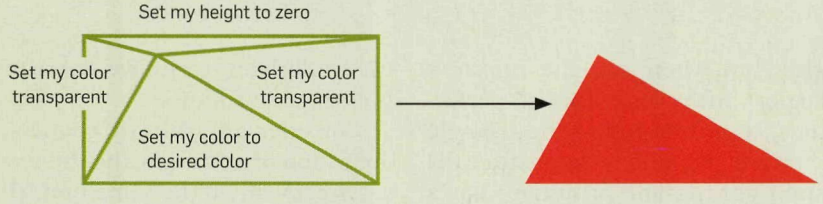


Figure 15. A small improvement.



into one right triangle (done) and one obtuse triangle (recurse on `drawTriangle`). Beautiful! (See Figure 13.)

To avoid infinite recursion in the third case, you must also stop if a triangle is too small—say, smaller than one pixel. From this description, all the corner cases are covered and any programmer should be able to write a correct implementation that performs well. When all the recursion has bottomed out, the decomposition of a triangle looks like Figure 14, which is what Lau drew in the first place.

His algorithm decomposes an arbitrary triangle into one or two triangles with horizontal legs; let's call those *aligned triangles*. It decomposes any acute aligned subtriangle into two aligned right triangles, and it decomposes any aligned obtuse subtriangle into a recursive number of aligned right triangles. Mathematically, this recursive number is infinite. Computationally, because you can render only a finite approximation of the mathematical structure on a screen with a finite pixel size, the number is logarithmic in the size of the figure.

Note the following small improvement: an acute aligned triangle can be rendered directly by setting the border opposite the aligned leg of the `<div>` to zero width and the borders on either side of the target triangle to transparent color, as in Figure 15.

Also note that it seems impossible to decompose an obtuse aligned triangle into a finite number of acute aligned triangles. Marc Levy provides the following argument sketch: if there were a finite number, then you could find the smallest one by sorting them by size. Consider the smallest one and draw a horizontal cut from its vertex farthest from the base of the original triangle. The residue is a triangle similar to the original triangle, thus leaving a smaller version of the original problem, in which there are even smaller acute aligned triangles. We assumed, however, that we had the smallest one, so that premise must be wrong: there does not exist a smallest one; therefore, there is not a finite number of them.

From a Teapot to Triangles and Back

Now that you know how to draw any triangle anywhere on the screen, how can you get the teapot? First, get some data from the public domain in a well-documented format called OFF (http://segeval.cs.princeton.edu/public/off_format.html) as seen in Table 1.

Table 1. Original teapot data; 3,751 triangles too much for good dynamic performance.

1976	3751	11253
-3.0000	1.65	0
-2.98711	1.65	-0.098438
-2.98538	1.56732	-0.049219
... 1976 vertex-coordinate triplets ...		
3 1455	1469	1459
3 1449	1455	1459
3 1462	1449	1459
... 3751 triangle patches as indices into the array of vertex-coordinate triplets ...		

Table 2. Teapot data after uniform decimation; 263 triangles with great performance.

166	263	0
0.000000	0.000000	0.488037
0.941342	-0.188153	0.626546
-0.000023	-0.032926	0.473905
... 166 vertex-coordinate triplets ...		
3 57	50	130
3 57	130	123
3 50	81	47
... 263 triangle patches as indices into the array of vertex-coordinate triplets ...		

Use a free graphics tool to decimate the data (trim it down to manageable size) to get the data in Table 2. Then convert the data into JavaScript via editor macros or a simple script. The final ingredients to the graphics pipeline are backface culling, Z-ordering, orientation by quaternions, and a kinematics library for spinning the object. Add a little spring-and-dashpot physics, and you can make your teapot bounce, morph, shatter, and unshatter. Code for all of these examples is available at https://github.com/gousiosg/teapots.

With the right leverage applied at exactly the right fulcrum point, it is relatively easy to do amazing things, such as rendering the classic teapot in HTML and CSS. Or, to paraphrase the Hacker's dictionary, we can extract great pleasure out of stretching the capabilities of programmable systems beyond the original intent of their designers. C

Related articles on queue.acm.org

Scripting Web Services Prototypes
Christopher Vincent
http://queue.acm.org/detail.cfm?id=640158

A Conversation with Ray Ozzie
http://queue.acm.org/detail.cfm?id=1105674

Mobile Application Development: Web vs. Native
Andre Charland, Brian LeRoux
http://queue.acm.org/detail.cfm?id=1968203

Brian Beckman (bbeckman@exchange.microsoft.com) is working with Bing on Maps and Signals and has held many positions at Microsoft since 1992, from Crypto (SET) to Biztalk to research in functional programming. He wrote the first version of the Time Warp Operating System on the Caltech Hypercube 1984–1989. He holds a Ph.D. in Astrophysics from Princeton University (1982) and has filed over 80 patents with 30 issued.

Erik Meijer (emeijer@microsoft.com) has been working on "democratizing the cloud" for the past 15 years. He is perhaps best known for his work on the Haskell, C#, and Visual Basic languages, targeting JavaScript as assembly language, and his contributions to LINQ and Rx (Reactive Framework). He is a part-time professor of cloud programming at TUDelft and runs the cloud programmability team at Microsoft.