Jim Blinn's Corner

Ten More Unsolved Problems in Computer Graphics ___

James F. Blinn

Microsoft Research

At this year's Siggraph I had the honor of delivering the keynote address in celebration of the 25th annual conference. In my talk I reminisced about my recollections of all the 25 conferences I had attended. I compared the state of computer graphics then with its state now. I predicted the future of the field. And I listed my choice of 10 unsolved problems in computer graphics. I have since realized that a keynote address is like a PhD thesis: After you create it, you can extract it chapter by chapter and publish it as several papers. Here, then, is the first extraction—10 unsolved problems.

History

Let's start by reviewing some past lists of unsolved problems. Judge for yourself how many of these historical problems have been solved by now.

Sutherland 1966[1]

The tradition of posing unsolved problems in computer graphics goes back, as most CG things do, to Ivan Sutherland. He started it all with a 1966 article in *Datamation* with the following:

1. Cheap machines with basic capability
2. Basic interaction techniques
3. Coupling simulations to their display
4. Describing motion
5. Continuous tone displays
6. Making structure of drawings explicit
7. Hidden line removal
8. Program instrumentation and visualization
9. Automatic placement of elements in network diagrams
10. Working with abstractions (scientific visualization)

Newell and Blinn 1977[2]

In 1977 Martin Newell and I presented the following thoughts at an ACM conference. The unsolved problems we focused on related primarily to realistic rendering. We were a bit lazy and could only think of six problems.

1. Increasing scene complexity
2. Fuzzy objects (hair, clouds)
3. Transparency and refraction
4. Extended light sources
5. Antialiasing
6. Systems integration

Heckbert 1987[3]

Paul Heckbert presented an update to unsolved rendering problems in 1987.

1. Converting implicit models to parametric
2. High-quality texture filtering
3. Antialiasing
4. Shadows without ray tracing
5. Practical ray tracing
6. Practical radiosity
7. Frame-to-frame coherence
8. Automating model culling
9. Smooth model transitions
10. Affordable real-time rendering hardware

Siggraph Panel 1991[4]

By 1991 things had become complex enough that it took a whole committee to identify key unsolved problems.

1. Managing scene complexity (Barr)
2. Tools for serious modeling (Brooks)
3. Large-scale user interfaces (Card)
4. Multimedia (Clark)
5. Automatic graphic design (Feiner)
6. Robust geometric algorithms (Forrest)
7. Better rendering (Hanrahan)
8. Graphics standards (van Dam)

When is a problem "solved"?

So now it's my turn again. But, to keep you on pins and needles, before I get into my list of unsolved problems I want to talk a bit about what "solved" means. Is a problem solved when its solution has been proved possible even though very expensive? Or does a true solution need to be cheap and easy to implement?

Some of the problems I decry below have, indeed, been solved in the theoretical sense. The problem remains unsolved in the practical sense, though, because cheap and fast solutions remain elusive. As you will see, many of the problems I present are more sociological and marketing issues than technical. Also, many of them have multiple parts with much overlap. I do, after all, have to come up with exactly 10 problems.

Blinn's Unsolved Problems

So, here they are, my own personal top 10.

1 Novelty

The first problem is simply finding something that hasn't been done yet. Let's face it; all the easy problems have been solved. Sometimes it seems that computer graphics research consists of finding some new subtle lighting effect that hasn't yet been modeled.

2 Education

There are two parts to this – learning and teaching.

In learning (keeping up with what has been done), not only do you have to find a problem that hasn't been solved, you have to *know* that it hasn't been solved. It used to be that the yearly Siggraph conference proceedings were the only place you could find computer graphics advances published. Now lots of places publish results. You can no longer keep up with every new development by reading only the Siggraph proceedings. Sometimes it's harder to discover an existing solution to a problem than it is to reinvent the solution yourself. There's lots of reinvention in this field; computer graphics is almost too easy in that regard.

In teaching (dissemination of new discoveries), just because somebody solves some problem doesn't mean that others will use that solution. This happens because other graphicists either aren't aware of the solution or they don't understand it. You can think of this as a marketing problem. Two examples that come to mind are premultiplied alpha (more about this below) and specular reflection calculation by raising the cosine of some angle to a power. I mean, Phong was a great guy and all, but his use of the cosine power was a simple approximation to a function that we dramatically improved on many years ago. Despite this, rendering systems still use "cosine power" as a property of surfaces as though it actually had physical meaning.

3 Systems integration

This is the problem of keeping all the balls in the air at once, that is, how to use all the tricks in one production. Just because one researcher can do cloth, one can do faces, and one can do hair doesn't mean that all animation systems can suddenly put them all together. The new technology of component software and plug-ins to existing software shows promise here.

4 Simplicity

I could keep this section simple and just say, "make things simple." But I won't, because life's not so simple.

How can we keep this stuff simple enough to use? Having a separate component for cloth, hair, skin, trees, water, physically based motion, deformations, texture synthesis, weathering, solid textures, multiresolution models, image-based rendering, yadda, yadda, yadda . . . could get a bit cumbersome.

But is simplicity even possible? Is it possible to make a simple computer graphics system that can generate complex images? Consider other systems that are complex enough to do interesting things. The telephone system has a conglomeration of old and new technology that still pretty much works. The human brain has several layers of legacy processors from our reptilian and mammalian ancestors. Maybe simplicity is a hopeless goal.

Nonetheless, you should still strive for simplicity. Let's face it, people don't read manuals any more. If a feature of a program is not obvious from playing with its user interface, then users assume that the feature either doesn't work or doesn't exist.

5 Better pixel arithmetic theory

Our basic concept of pixels as red, green, blue, and alpha channels is incomplete. I can see at least three main problems. One is largely a matter of definition and education, while the other two involve integration of compositing with other parts of the imaging process.

Problem one concerns premultiplication of the color channels by the alpha channel. We've long known that premultiplication has many advantages. For example, it allows compositing arithmetic on all channels to be identical, linear filtering to commute with compositing, and the set of pixel values to be closed under compositing (This is because the result of any pixel compositing operation is a premultiplied pixel. Therefore, the first operation in a chain of operations moves us into the premultiplied domain automatically.) Nonetheless, several systems store colors unmultiplied by alpha. This indicates a fundamentally different interpretation of the meaning of the alpha channel. We need to understand this difference and realize that the alpha value has subtly different meanings when used as a fundamental component of a pixel and when used as a stencil to shape an existing image. I call this the local/global alpha distinction. Both uses of alpha are important, and systems should support both.

The second problem is that the conventional alpha channel interpretation assumes that edges of a foreground and background object are uncorrelated. Some useful algorithms, on the other hand, divide the screen into nonoverlapping polygonal regions. When two adjacent regions are rasterized, their boundaries coincide, of course, and thus are completely correlated. These rasterized regions cannot be merged using the standard uncorrelated compositing algebra.

Thirdly, we must consider combining compositing operations with light reflection models. The fundamental operation of light reflection is the simulation of colored light reflecting off a colored surface or transmitting through colored glass. A single alpha channel can only model partial geometric occlusion of a pixel. It cannot adequately simulate a colored, partially transparent surface. A separate alpha channel per color still isn't the ultimate answer, either. The complete physical simulation of spectral interactions seems necessary, but might be overkill. In addition, other lighting-related arithmetic operations that must be included include the simulation of after-the-fact shadow application and transparency effects at boundaries of fogged objects.

A complete algebra on pixel values will be embedded in the deep innards of any rendering system. Currently available compositing operations do not address the above concerns. A more complete theory must be devised and installed in the inner polygon tiling loops of future 3D APIs.

6 Legacy compatibility

Time goes on, and we do things differently. Partly this occurs because we have learned how to do things better than we did before, and partly because technological improvements change the trade-offs to make things practical that weren't before. Unfortunately, our history remains to haunt us in the form of legacy applications and data. This applies to operating systems, 3D APIs, file formats, and so forth. Simply pitching out all legacy items isn't a good idea. Instead, progress is a balancing act of how to not abandon the old, while allowing the new.

One particularly interesting example of this is the coming convergence (or collision) of television technology with computer imaging. We will have to face the fact that TV pixels (even digital TV pixels) are not the same as computer pixels. For one thing, TV pixels have a different range than computer pixels. For digital TV the byte value 16 corresponds to black, and the byte value 235 corresponds to white (220 levels total.) For another, TV pixel values do not linearly represent light intensity; they have a gamma correction value burned into them. Sometimes computer graphics pixels do this, and sometimes they don't. Gamma correction has the advantage of giving better resolution to darker regions of an image, but doing correct image compositing with it is slow. The probable eventual solution to this problem is to convert everything to use 16 bits per color channel and encode linearly. This will have roughly equivalent dark resolution to 8-bit gamma corrected pixels. We still need to work out the best scaling within this range, though, and to allow small negative values and greater-than-one values.

7 Arithmetic sloppiness

We're doing a lot of things wrong in image rendering, and we know it. We do it anyway, though, to appease the great god of speed. Surprisingly, this problem worsens because modern computers are so fast—just fast enough that some algorithms are borderline real time. Programmers are tempted to do a sloppy job of pixel arithmetic to get their speed just over the line into real time. This can often lead to an embarrassing amount of arithmetic sloppiness in pixel calculations. I'll mention a couple of examples of this:

- When processing pixel values, we usually ignore any of the above-mentioned nonlinear gamma encoding—we simply do linear calculations on this nonlinear data. I discussed this problem in more detail in my January/February 1998 column, "A Ghost in a Snowstorm" (pp. 79-84).
- Conversions between, say, 5 bits per color channel and 8 bits per color channel proves trickier than most people realize. Proper conversion isn't a simple 3-bit shift and mask operation. A 5-bit quantity is a number of 1/31 st's, an 8-bit quantity is a number of 1/255 th's. Proper conversion requires multiplying or dividing by the quantity $255/31 \approx 8.22$.
- Texture filtering often takes the form of simple bilinear interpolation between the four nearest texels to the desired pixel. This bad interpolation adds ugly diamond-shaped artifacts to the image.

The problem with all these picky details is that often the proponents of bad arithmetic show images that are, visually, pretty much the same as correct ones. The bottom line is that we need better criteria for just how accurate we need to be.

8 Antialiasing

At the turn of this century the director of the US Patent Office stated that all possible inventions had already been invented. Bill Gates is often quoted as having said that 640K of memory is enough for anybody (he was right, too). I think I can become famous, too, by categorically stating what will *not* happen (thus guaranteeing that it *will* happen). I therefore proclaim that nobody will ever solve the antialiasing problem. It does, you will note, appear twice on our list of historical unsolved problems. No one will ever figure out how to quickly render legible antialiased text in perspective. Textures in perspective will always be either too fuzzy or too jaggy. No one will ever build texture-mapping hardware that uses a 4×4 interpolation kernel or anisotropic filtering. And no one will ever send me tickets to the Digital Domain Siggraph party.

9 A modeling, rendering, animation challenge

OK. So much for the hard sociological problems. How about a simple straightforward rendering challenge? Here it is:

Spaghetti.

No, really. Consider that we can do cloth pretty well now (possibly due to our obsession with rendering the human body). We can model how it drapes and folds without self-intersections. Now cloth is a basically two-dimensional shape. Spaghetti is an essentially one-dimensional shape. It should be even easier to model. Such algorithms could also apply to piles of rope or string and even conceivably to protein folding.

And remember, to respond to this challenge you must solve all three problems: modeling (shape), rendering (making pictures), and animation (showing evolution over time). Don't forget the sauce.

This will give new meaning to the term spaghetti code.

10 Finding a use for real-time 3D

We all know that real-time 3D is cool. And what's incredibly cool, and astonishing to us old timers, is the fact that you can now get real-time 3D hardware for about a $100. What's not cool is that the companies making these hardware cards are having a tough time staying solvent. The main applications for cheap 3D hardware, games, simply don't have enough adherents to support the industry. To keep 3D hardware cheap, we need more large-scale uses for it. And I mean *large* scale, uses that virtually everybody owning a personal computer will lust after. Fruitful areas might include electronic commerce and business data visualization.

Here's another idea: a vision of better 3D user interfaces. Currently, operating systems and applications have a lot of persistent settings that indicate preferences and system setup information. In order to examine and change these settings, you have to hunt around through

a maze of windows and menus to find the particular one that applies. Suppose we could represent this system state in terms of 3D shapes rather than list settings. The internal state of your program would then look something like an old-fashioned car engine (one simple enough to understand, I mean). You would see interrelations between components as shapes plugged into the "system" shape. Direct manipulation of these shapes via a mouse or data glove would make configuring your system a much more understandable process. (I am reminded of the scene from the movie *Johnny Mnemonic*. . . .)

Get hopping

I realize that people will mostly attack the easiest of these: spaghetti. The other problems will likely require group participation but are, probably, rather more important. Whichever challenges you—hop to it! And let me know what you find. ∎

References

1. I.E. Sutherland, "Ten Unsolved Problems in Computer Graphics," *Datamation*, Vol. 12, No. 5, May 1966, pp. 22-27.
2. M. Newell and J. Blinn, "The Progression of Realism in Computer Generated Images," *ACM 77 Proc.*, Oct. 1977, pp. 444-448.
3. P. Heckbert, "Ten Unsolved Problems in Rendering," *Workshop on Rendering Algorithms and Systems*, Graphics Interface 87, April 1987, Toronto, http://www.cs.cmu.edu/afs/cs/user/ph/www/unsolved.ps.Z.
4. "Computer Graphics: More Unsolved Problems," Siggraph Panels 1991, http://www.siggraph.org/publications/panels/siggraph91/p04.html.

Contact Blinn by e-mail at blinn@microsoft.com.