

# Tool Postmortem: The Shockwave3D Engine from Intel and Macromedia

## Introduction

In April of 2001, Macromedia released Shockwave/Director 8.5. Shockwave, of course, is the reigning king of interactive, rich-media web content players, and Director is its parent product, a powerful multimedia composition studio. This latest version of the venerable player and its authoring tool heralded their transition into the world of web 3D with the inclusion of a full-featured 3D engine. A group within Intel Labs dedicated to 3D technologies developed this engine, referred to as Shockwave3D. The story of this engine's development consists of equal parts plague and panacea. We made as many poor decisions as we did clever ones, had as much bad luck as good, and in the end worked hard enough to tip the balance.

## Motivation

At Intel, our group is chartered to research and realize technologies for managing, simulating, and rendering 3D worlds. In the past, the group's work has manifested in the form of Digimation's Real-Time (RT) 3D Libraries, and the [MultiRes plugin](#) for 3ds max. In early 1999, the group began discussing the idea of doing something closer to a full-fledged engine to both leverage and support the technologies being developed internally. Basically, the group had developed a number of robust, loosely collaborative, 3D technologies which we felt weren't getting as much daylight as they deserved, primarily because we didn't have a nice package for them. Furthermore, the team was interested in pursuing ideas that required a more formal component infrastructure and the presence of a legitimate scene graph hierarchy. The incentive quickly morphed into an opportunity to give web 3D a kick in the pants.

We had neither the resources nor the justification to develop, in addition to the engine, a full application framework and authoring tool, without which the engine isn't very attractive as stand-alone technology. Thus, as the idea took hold, we began to search for a distribution partner. Macromedia was the obvious choice not simply because of the wide acceptance of their player but because its capabilities and feature-set were deep and mature, making it a good candidate to not just survive the leap to 3D, but to really take advantage of it. Intel engaged Macromedia during the second quarter of 1999, with the design effort already underway. The ink didn't dry on the contract until early 2000, by which time both companies had already rolled up their sleeves and were forging full steam ahead.



A 3D scene exported from Maya and loaded into Director. [View it online.](#)

## What Went Right

### 1. Relationship with Macromedia

The decision to partner with Macromedia was probably the best thing that happened to the project, for two independent reasons. The first reason was the incredibly compatible goals of the two companies. It was just a great match. Intel's vision for a web 3D engine was very consistent with Macromedia's vision for introducing 3D capability into Director and Shockwave. The timing was also very compatible. That's not to say that the planets were aligned, but Macromedia already had dreams of 3D sugarplums dancing in their heads when we approached them. And it wasn't at all difficult to get the estimated development effort to synchronize with Macromedia's estimated release cycle.

The fact that we were able to weather the severe turbulence that would manifest in the latter phases of development was due in no small part to the high degree of alignment between the two companies. Had we not been in such a clear win-win relationship, either company could've softened their resolve, resulting in a very weak product.

The second reason why the relationship with Macromedia was such a positive was the level of communication and cooperation that was nurtured between the engineers on both teams. It didn't matter how well our companies' goals lined up, a sour environment between the two groups of developers would've eventually scuttled our success. The relationship was strained at times, to be sure, but the fact that everyone worked hard to maintain honest, respectful, professional attitudes enabled us to react intelligently to our mistakes, and to be objective when tough compromises were on the table. The managers at both ends also did the right thing, allowing relatively unfettered communication between the two teams. Our ability to resolve both technical and logistical issues would've been significantly crippled had we not been allowed to communicate directly and frequently with the Macromedia developers, not to mention the fact that the camaraderie I mentioned would likely never have taken root in the first place.

This was the first time our group had pursued a cross-company project with such tightly interwoven responsibilities. It could've easily been an unmitigated disaster. Both development teams were in uncharted territory in many different ways. The compatibility of our goals with Macromedia's, and the positive attitude maintained by the two teams were both instrumental in our ability to overcome what at times felt like a heavily stacked deck.

## 2. Ownership of Integration Implementation

A significant part of development was the need to extend the Director/Shockwave scripting language, Lingo, to expose the 3D engine. All of our core code was developed using internal interfaces, components and data structures, collectively referred to as IFX. The soft underbelly of Director and Shockwave is exposed via its own set of interfaces, components and data structures, referred to as MOA (Macromedia Object Architecture). In order for the 3D engine to function as a seamless part of Shockwave, we needed to create a layer that would arbitrate between IFX and MOA, in effect extending MOA, and therefore Lingo, to support 3D. In a fit of creativity, we decided to call this arbitration layer MOA3D. Originally, it was decided that we would craft the new MOA3D interface definitions, and that Macromedia would be responsible for the actual component implementation that would support those interfaces and enable the new 3D Lingo syntax. I'm not sure what made that sound like a good idea at the time, because it wasn't.

It didn't take long for us to realize that not being responsible for integration implementation was a very bad thing. Director was the only environment we had in which to instantiate the engine and test functionality. If the MOA3D implementation didn't stay one step ahead of the engine development, then our team was forced to either check in code that hadn't been fully validated, or not check it in at all and wait for the MOA3D support to catch up. But there was an even greater fundamental flaw in the proposed division of labor. The individual engineers on our team needed to be able to exercise new code in the product environment, including the new Lingo syntax, AS they developed, not after. Most developers hit peak productivity when they're in a very tight code-compile-test loop. Since we weren't supposed to be implementing the MOA3D code, we basically had lost control over 1/3 of our productivity cycle. The developers needed handles hooked up to new features in order to test them iteratively, but they had to wait for someone else to come along to attach the handles. This situation probably festered much longer than it should have, but eventually the inevitable happened and we just started assuming responsibility for the MOA3D implementation. In addition to removing a large speed bump from the development road, this ad hoc ownership of the integration implementation paid big dividends later, as we were able to deal fairly gracefully with the chronic interface and specification churn that infected the post-beta stages of the product.

## 3. Render Layer Abstraction

In order to describe this section, I must first elucidate the nature of our engine's rendering support. The engine supports four renderers: software and OpenGL 1.1 on Win32 and Macintosh, with DX 5.2 and DX 7.0 also supported on Win32. Most of the engine is blissfully ignorant of these four APIs thanks to a nice, clean, rendering abstraction layer.

For the record, this abstraction layer was one of the few things that went according to plan and worked (almost) flawlessly. Granted, we had to sacrifice an engineer's sanity for it, but it was worth it! The render layer interfaces matured early, and although our sacrificial engineer Dan Johnston spent the rest of the project cursing virtually every driver and graphics card in existence, the rest of us were free to go about our business in the engine, quietly thinking "better him than me." Now when I say that it worked almost flawlessly, I don't mean to imply that it didn't have any bugs. In fact, if we looked at a pie chart of the bugs, I suspect the render layer would get the biggest slice, but they were bugs inside the abstraction layer, mostly resulting from driver irregularities. I can say the render layer performed fabulously because its job was to prevent the developers from having to deal with any of the complexities associated with simultaneously supporting four different rendering APIs, and that's exactly what it did.



Nice looking cave fly-through. [View it Online.](#)

#### 4. Partnership with NxView and Havok

Early in development, we recognized the need for a much stronger software renderer. We'd been relying on the DX5.2 software renderer, but knew that Macromedia wouldn't be willing to leave Mac users out in the cold if they didn't have 3D hardware. Without going into details (because I don't know them) I'll just skip to the end and say that Macromedia licensed the software renderer from [NxView](#). Now, this is one of those issues that involve some speculation. I'm including it among the things that went right only because I'm speculating that without it, things would have been much worse. There were some frustrating side effects of using the licensed renderer, most notably the inability to address bugs or feature inconsistencies in a timely manner because we didn't have access to the source. Still, the general consensus, with strong endorsement from our team's rendering API guru Dan, is that getting the NxView renderer was a good thing. We had our hands full (and by 'we' I mean 'Dan') trying to reconcile the quirks of each API with the abstraction layer, as well as the plethora of esoteric bugs resulting from the myriad of hardware/driver/API combos. Had there been the additional burden of actually developing a fully-featured software renderer from scratch, we likely would've had to cut corners in very undesirable ways.

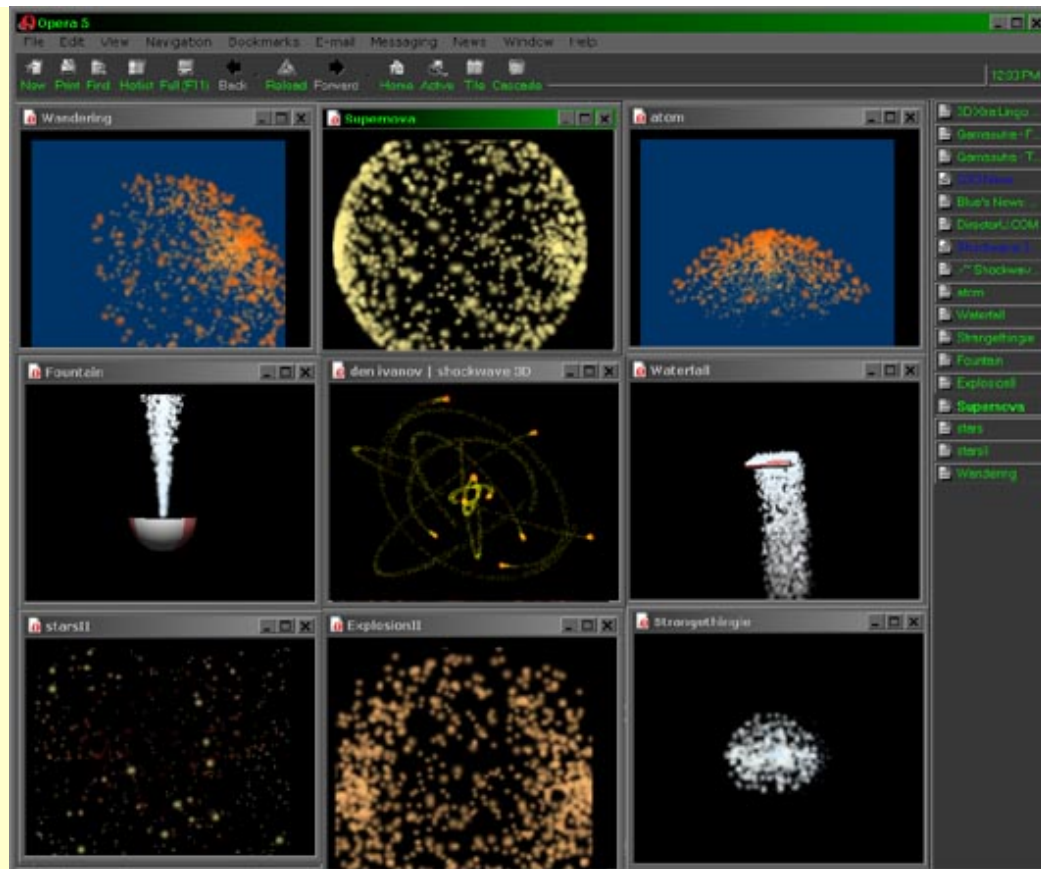
It's worth noting that a software renderer was not an optional feature. Macromedia was very adamant regarding the minimum hardware requirements for the engine, and 3D graphics hardware was not among them. It's been a few years now that game developers have had the luxury of requiring 3D graphics hardware for their games. They do it now without batting an eye. But we had to step back in time a little bit to ensure that we didn't leave a significant portion of Macromedia's Shockwave users behind. Suffice to say that we had to provide a software renderer, and that NxView offered the most convenient way to do it. And it's a very nice renderer, to boot.

The second partnership that was a huge positive was the one brokered with [Havok](#), enabling the engine to include very fast, robust support for full rigid-body physics. The requirement for physics support surfaced well after development began, and we actually sacrificed another engineer to develop an in-house physics system from the ground up. The investment we put into the in-house solution was huge, and yielded a very respectable physics engine, with typical limitations. We communicated these

limitations to Macromedia, who deemed them unacceptable. At this point, we began negotiating with Havok to figure out how to incorporate their rigid-body physics system into Shockwave3D. Because we were in a late stage of development when this issue came to a head, that integration wound up being very loose, and we sacrificed some performance as a result. Still, it worked fine, and we yanked out our own physics system from the engine. The interesting part of this story is that Havok's limitations are mostly identical to ours, so we didn't really gain anything in that respect. What we did get, however, was a physics solution that was much more mature, robust, tested and optimized than we would have been able to provide in our nascent solution. Like the acquisition of the software renderer, Havok allowed us to concentrate on other core features, ultimately giving us better bang for our buck.

## 5. Advanced Features

One of the issues that caused chronic discontent during development was the difference of opinion between the two development teams regarding the level of complexity that should be allowed to exist in the exposed features of the engine. Macromedia, quite understandably, was reluctant to take the fire hose of 3D and point it straight at their loyal Shockwave developers. On the other side of the fence, we were very reluctant to arbitrarily eliminate the ability to do things with the engine just because they were complicated. We felt that we should give Shockwave developers the opportunity to use the engine in whatever way it was capable. The Shockwave developer doesn't *have* to use every feature, we argued, and if we take care to keep the default settings intuitive, they can learn how to use the engine in small steps. "They'll grow into it," was essentially our argument. Fortunately, this argument succeeded in winning a compromise. Admittedly, much of the complexity and flexibility in the engine remains unexposed, but some very powerful features that were in danger of getting cut altogether were allowed to survive. Most notably among these are the particle system, and the bones-based animation. As a surprise to everyone, in fact, the particle system has been one of the most widely used features of Shockwave3D by Shockwave developers teaching themselves how to use the engine. This isn't meant to imply that Macromedia was wrong. They were completely correct in their assertion that the more features there are, the more intimidating is the prospect of learning how to use the product. Hopefully, we were able to strike a reasonable balance. Only time will tell. If it only takes a Shockwave developer a few months before she starts lamenting the inability to do this or that, then we've erred on the side of simplicity. If we don't hear any lamenting because there are only a handful of Shockwave developers using 3D, then obviously we've left the curve too steep.



Fun with particles. [Try one.](#)

## What Went Wrong

We certainly didn't break any new ground in the "What Went Wrong" department, committing most of the Cardinal Sins: insufficient design and documentation, naive scheduling, specification flux, and short shrifted QA. I've heard some argue that these things are going to rear their heads in every project, no matter what you do, so maybe we can take some solace in the idea that we erred in ways that were unavoidable. I don't believe that, personally, but if nothing else, there is some reconciliation to be had in the fact that these are all obviously weeds with very deep roots.

### 1. Distribution of Knowledge

Vision, documentation, communication, distribution of responsibilities; we didn't discriminate against any of them. We mismanaged all with equal measure. We broke ground very early on the hole that we would dig for ourselves by designing the initial architecture in a vacuum. I mentioned at the beginning that the primary motivation for the engine was to provide a framework for the team's existing set of 3D technologies, which included Multi-Resolution Meshes (MRM), subdivision surfaces, a complete animation system using both keyframe and bones-based animation, non-photorealistic rendering, and a robust particle system. The problem was that these technologies, and specifically their data dependencies, were not described in detail and factored into the early design. This wasn't necessarily a conscious decision, but neither was it an accident. I believe the expectation was that the technologies were modular enough to allow them to simply adopt a few new interfaces and then get bolted into the architecture. In fact, the engineers originally responsible for the technologies weren't slated to help develop the Shockwave3D engine. A subset of the team was going to do the engine, allowing the remaining team members to continue to pursue new research projects. Of course, this didn't happen. Eventually it became obvious that a much higher degree of involvement was required to get all of the technologies not only integrated but cooperating, at which point the rest of the team got pulled into

development. Thus, our first example of poor distribution of knowledge was the failure to involve the entire group in the design process.

The repercussions of this first misstep could have been mitigated had there been diligent documentation of the architecture. Instead, documentation in the project was neglected early and often. When the number of people working on the engine effectively doubled, there were insufficient and outdated design documents to aid in their assimilation of the engine's complexity. That knowledge was held collectively in the heads of essentially two people. The most relevant part of the architecture that wasn't documented was the nature of the resource sharing that occurs in the engine. Scene entities can participate in resource sharing at many levels, including geometry, shaders, material properties, and geometry modifiers.

Because these complexities weren't captured anywhere, as more and more people got their fingers in the code, changes were often made by one or several engineers that would handicap a seemingly unrelated part of the engine, in subtle (and sometimes not so subtle) ways. These issues would typically be discovered not simply after the fact, but well after the fact because full regression testing wasn't occurring until very late in the development cycle. At that point, other dependencies might have grown around the original change and it became increasingly difficult to resolve the architectural problems. The fallout of this process was the deteriorating coherency of the code and a growing collection of fixes whose nature and origin were only fully understood to one or two engineers. In other words, the whole thing began to feed on itself. Poorly distributed knowledge of the system led to bugs, which in turn led to shortsighted fixes that weren't widely understood.

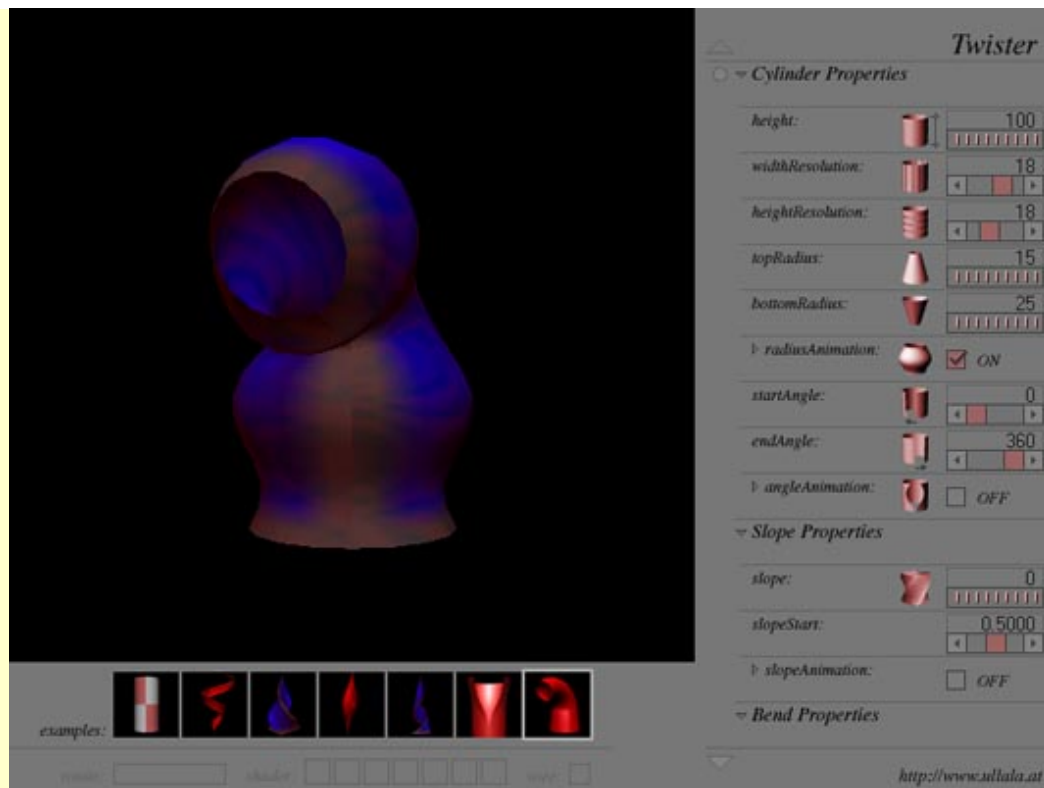
The last bit of laundry to air regarding distribution of knowledge is that we didn't do a good job reinforcing the project vision, particularly as the team grew in size. Because the vision was not clearly communicated, the large development team wound up designing to goals on an individual basis. These disparate goals introduced inconsistencies. Each engineer couldn't refer to a strong overriding project vision when faced with tradeoff decisions, such as speed vs. size, simplicity vs. control, etc., or even the rudimentary process of building confidence in their day-to-day direction.

## 2. Scheduling

Our problem wasn't that we didn't schedule enough time, although that turned out to be the case, but rather that when the schedule needed to be adjusted, it was micro-adjusted, again and again. Had we not committed all of the other mistakes described here, the original schedule would have been reasonable. In other words, I don't think the scope of work required for the engine was completely misjudged. The schedule failed not because of the technology; it failed because of our processes. But again, our mistake was not in the original schedule, it was in how we reacted to the need to adjust the schedule.

Our productivity was negatively affected by the constant application of micro-adjustments to the schedule and deadlines. Instead of recognizing the need to adjust the deadlines dramatically and then doing so, we just nickel and dime'd the schedule to death, and succeeded only in smothering ourselves under a relentless barrage of petty deadlines. As programmers, this prevented us from hitting our full-stride because there was constantly the spectre of yet another deadline just around the corner that would cripple decision-making and shackle efforts to implement the appropriate solutions to problems. This is largely why the implementation problems described in the previous section were often addressed in an unsatisfactory manner. Often we would identify what we'd done wrong, identify the architectural implications, and how to properly resolve the issue, only to be unable to apply the solution because the frequency of the deadlines prevented us from taking the one step back needed to go forward. We've since joked that we were under an effective code freeze for 12 months.

Even more detrimental to our sanity, however, was our unbridled compulsion to reuse the time associated with each incremental schedule shift. By that I mean that instead of using the additional time made available by a schedule shift for addressing whatever caused the shift in the first place, we treated it like an opportunity to squeeze more functionality into the engine. This wasn't some subversive rebellion by the developers, or a misguided mandate from the team leads or managers, but just a lack of discipline and foresight by all of us.



Experimenting with procedural mesh deformation. [Play with it!](#)

### 3. QA and Content

Two separate but inter-related areas negatively affected us: the resources committed to QA and the resources committed to content creation. The inter-relationship between these two areas was heightened because of the fact that we were developing a tool to enable content as opposed to content itself, as is the case for most game developers. Thus, content creation during development was serving the same goals as the directed QA effort, as opposed to being an end unto itself.

Our QA department was hampered by too much turnover and too little automation. The turnover is inherent in our QA department because everyone except the QA managers are in contracted positions. Furthermore, the maximum length of a contract is one year. This is a sufficient amount of time to retain QA personnel for most projects within the Intel Labs. Our project, however, was of a greater size and duration than is normal and the QA department was not structured to accommodate it. The lack of automation is a direct result of the enforced turnover, and again is typically not an issue for projects in the Labs. The scope of the functionality within the Shockwave3D engine, however, created an intense need for automated testing procedures and we never provided them.

We didn't accompany the development of the engine with sufficient complex content creation. In effect, it's like we were building a racecar but not ever racing it. This neglect for scenario testing, when compounded with the high turnover in our QA ranks, led to a very poor distribution of defect discovery: lots of easy bugs early and hard bugs late. There's an interview somewhere of Carmack where he is asked if id has ever considered just licensing technology and not actually developing games. He responds that such a situation would never work because of the fundamental difference between the ways the engine gets used by developers versus artists (and level designers). Artists don't know what kinds of things the engine can't do, so they just do them. We can attest to this. For a while, we had an outstanding intern artist developing content for us, and he consistently did things that he wasn't "supposed" to do, exposing all manner of flaws, usability issues, and inconsistencies. Ultimately, though, it turns out that Carmack was exaggerating. Developing an engine without simultaneously developing a game can work; you just have to make two dozen programmers miserable for 18 months to do it.

#### **4. The Exporter SDK**

At some point, our group went from being responsible for developing a Shockwave3D exporter for 3D Studio Max to being responsible for an exporter AND a multi-platform SDK. We never correctly compensated for this change, and the effort was under water for the rest of the project. I'm going to rattle off a number of issues that cast a shadow on our exporter/SDK development, but they're pretty much all related to the fact that we didn't recognize the degree to which we were short-shrifting the effort. Another major handicap for the exporter/SDK was the problem I'll discuss below regarding the ever-changing product specification. Third party 3D tool developers were unable to really start work on their exporters until around our fourth beta release, since the SDK consisted of pretty much just the raw IFX interfaces, and those took much too long to bake. Furthermore, those interfaces represent a myriad of styles, failing to adhere to a uniform design standard. While I'm on the topic of the interfaces, I also have to mention that parts of the IFX API that made up the SDK wound up not being fully or intuitively functional, mostly due to dead weight from interface drift.

We never actually created a complete design for the Shockwave3D Max exporter. Its functionality grew whenever some engine feature got hot and needed content. As a result, it's role as a reference implementation is diminished. Ultimately, the fact that Alias|Wavefront, SOFTIMAGE, NewTek, Caligari, etc. have been able to develop excellent Shockwave3D exporters is a testament to their patience and dedication, and not because we made it easy for them.

#### **5. Specification Flux**

The product specification seemed to subscribe to a motto of "change late, change often". We were plagued by incessant adjustments to the spec, almost all the way up to release. To some extent, the transient nature of the specification can be considered both a result of and a contributor to each of the other major things that went wrong. It wrecked havoc on the QA efforts, causing chronic re-implementation of the unit tests. When faced with a non-fatal bug, we learned to first ask if it was really a bug or just an outdated expectation. The spec changes also caused tremendous grief for the several brave souls dedicated to the exporter and SDK. Generally speaking, though, the constantly fluctuating requirements led to churn in our interfaces, and subsequent pollution of data management and abstractions in the engine. The instability really started to wear on the developers, infecting us with the jaded impression that no matter what we implemented, chances were we'd have to go back and implement it again.



A simple scene used to good effect. [View it.](#)

### The Bottom Line

Here's where I get to say that despite all of the turmoil, all of the miscommunication, all of the scheduling gaffes, miss-prioritization, and poor resource allocation, the product turned out just fine because everybody on the development team rocks.

While we missed the mark for highly complex, large game worlds, the engine is quite adept at virtually everything else. And that's not to say that complex scenes with a high model count can't be achieved with Shockwave3D, only that such scenes impose a far greater micro-management burden on the Shockwave developer than we'd intended. What the engine does, it does extremely well, but we compromised some of both our design and implementation goals to get there. The engine is close to what we envisioned, though the implementation is a little too disheveled to give us much opportunity for reuse. We will need a clean slate if we're going to significantly grow the capabilities and performance. I feel as though the engine ended up as capable as it is because we had such lofty goals in the first place. As ironic as it sounds, I also think that the engine's saving grace is that it had such a broad target for applicability. Because we were aiming for such a tremendous amount of flexibility, the extensions to Lingo (the Shockwave scripting language) were done in a way that really accommodates future growth. So although we may have burned some bridges with the implementation, much of the design and scripting syntax afford us the opportunity to really ratchet up the capabilities of the engine in the future, without compromising the knowledge base currently being cultivated among Shockwave developers.

Ultimately, We were able to pull together a very unique set of features, in a way that's accessible to novice 3D programmers like nothing else out there. We're starting to see Shockwave3D fulfill its potential as a tool for affordably and quickly exploring 3D UI design and gameplay mechanics. PC and console game developers are also beginning to use Shockwave3D for showcasing models, animations and other game content on the web. Most importantly, though, we're witnessing the transformation of Shockwave developers into legitimate 3D programmers. I have every confidence, and mounting evidence, that they'll outgrow the training wheels currently in Shockwave3D and demand not only more sophisticated features but also undiluted control over the entire engine. The ease with which they're

able to use Shockwave to experiment with 3D has catapulted them over the learning curve and given them a taste for the potential of interactive 3D on the web.

## Summary

Company:	Intel Corporation
Full-Time Developers:	23 programmers, 6 QA (not including Macromedia personnel)
Release Date:	April 2001
Platforms:	Windows 98/NT/2000, Mac OS 8.5-9.1
Critical Dev Hardware:	500MHz-1.4GHz P3/P4, Mac G4, a ton of graphics cards.
Software Used:	<a href="#">Director/Shockwave</a> , <a href="#">Perforce</a> , <a href="#">MSVC++ 6.0</a> , <a href="#">CodeWarrior</a> , <a href="#">VTune</a> , <a href="#">Intel Compiler</a> , <a href="#">3DS Max 3.1</a> , <a href="#">Character Studio 2.5</a> , <a href="#">PERL</a> , <a href="#">xat.com</a> , <a href="#">WebSpeed Simulator v1.5</a> , <a href="#">PVCS Tracker</a>
Notable Technologies:	Multi-Resolution Mesh, Subdivision Surfaces, Skeletal Animation, Non-Photorealistic Rendering, Particle Systems
Lines of Code:	~343K (420K w/ headers)

## Author Bio

David Hostetler has been a software engineer in Intel's Graphics and 3D Technologies group for the last 2 years. Prior to that he was in school for reasons that now elude him.

## Links

### Shockwave3D Development Info

[3D Tool Vendors & Shockwave3D](#)  
[Shockwave3D SDK](#)

## Demos

[Cool demos](#)  
[More cool demos](#)  
[Havok demos](#)

## Shockwave3D Tutorial Sites

[www.directordev.com](http://www.directordev.com) (cookie required)

[Director University](#)

[Director-3D](#) (a dedicated 3D forum)

[Director Online](#)