Design of a Modern Image Generation Engine for Driving Simulation

Bob Kuehne, Sean Carmody
Blue Newt Software, LLC
201 South Main St. Suite 503, Ann Arbor, MI 48103
rpk@blue-newt.com, sean.carmody@blue-newt.com

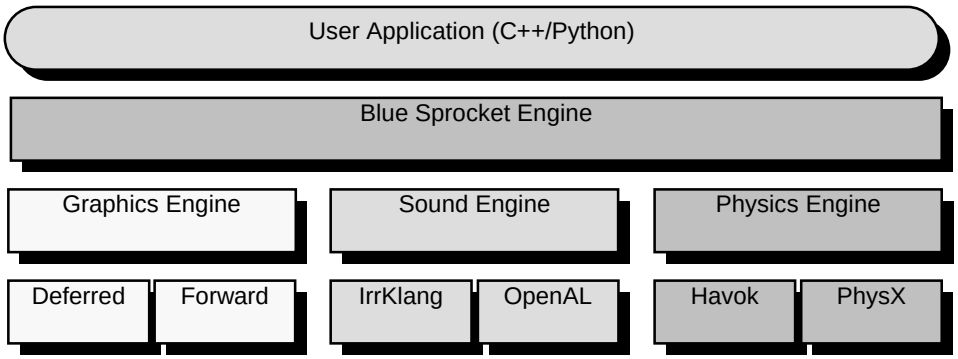
Abstract *– In 2007 Blue Newt Software began designing and building a new visual rendering system for visual simulation markets. The image generator, called PixelTransit, is built on top of our engine called Blue Sprocket. Our rendering technology has been instrumental in demonstrating to new customers how graphics hardware can be used not just to create better images, but to gain better insight into their simulated environments. Blue Sprocket was designed to address four core goals from a rendering perspective: improved performance, higher-quality, scalability, and improved lighting. The engine additionally was redesigned to use standards wherever possible, and to bring a degree of modularity and scalability not available in this domain. This paper will describe the design and implementation of this system and discuss current problems and future work to be done in this space.*

Design

We built our rendering technology having analyzed and visualization requirements with many different customers. A common refrain was that as new technologies for rendering emerged, customers wanted these advanced graphics features, but were unable to easily adopt them. These new technologies might be newer scenegraphs with different rendering techniques, updates to standard application programming interfaces (APIs) such as OpenGL, or simply the latest algorithms and research from SIGGRAPH. Regardless of the source, adopting these technologies was something that each company addressed every few years to move their simulators forward. We decided a new approach was necessary to help our customers focus on their domains and let us continuously refine, revise, and provide new techniques and algorithms for rendering. We do this through a design which integrates both commercial and open software components, on all platforms. In short, our strategy is to create long-lived interfaces, but with flexible underpinnings allowing future enhancement.

Architecture

We designed and built the Blue Sprocket Engine (BSE), our rendering and simulation software development kit(SDK) by researching the needs of a variety of customers in this space and studying the many open-source games APIs available. We combined our experience with the simulation market with our research to create an architecture that would allow rapid innovation with stable interfaces. We created a component-object model in which the simulation end-user would create logical objects representing simulation entities and their articulation, then attach components, or 'viewable' aspects to those objects. Viewable components are principally visual, such as 3D models, but can also be sounds, physics, etc. We provide interfaces to the most common of these including sound, physics, and of course, our OpenGL-based graphics. These components are then assigned by the developer to a processing entity we call an Engine. An engine is a specialized processor for turning components into some output, typically imagery, sound, or updated state of the world, as in physics processing. The diagram below shows this overall architecture.



The key architectural decision we made early was to create a simulation object API in the Blue Sprocket Engine to insulate users from changes lower in the API stack. We then were able to create separable instances of Engines which accomplished rendering via various mechanisms. For example, users with the same simulation software can choose to run in a forward rendering environment for ultimate speed or a deferred shading environment for lighting with unlimited numbers of lights. That choice, however, can be made based on the needs of the particular simulation run, and does not constrain the user to only developing their application for one or the other. Mitigating the pain of moving from one rendering interface to another was the key goal of the design of the Blue Sprocket Engine.

Engines are the workhorse of our system, and are individually responsible for two components of how a user creates a simulation. First, they provide components, which are attached to the user Object hierarchy that they describe their simulation world with. Second, engines provide processing capabilities to turn the components that they're managing into some coherent view. The most common view of this is the Graphics Engine which turns Graphics Components containing geometry and rendering state into an image.

Platform

We began developing our technology with the choice of hardware platform on which to deploy. Based on our experience, we knew many customers had existing Windows installations, however, we wanted to move beyond Windows to ensure both higher code quality, and preserve options for our customers. That decision meant that our tools and code had to be fully cross-platform. This choice of platform directly leads to decisions about which technologies we integrate. Today we build and deploy on 64-bit Windows, Linux, and Mac OS X[7]. This lets our developers and customers both work where they're most productive. Beyond that simple business necessity, we also catch many potential problems early due to compiler differences among vendors.

Technologies

Our core product focus is clear: to choose and integrate technologies that provide our customers value, while giving those customers programming interfaces which will remain stable, over a highly flexible and high-performance rendering core which can be used to build engines for today's and tomorrow's hardware platforms. This guiding principle informs how we choose among technologies to integrate, build-upon, and deploy.

Given limited resources with which to develop a product, we're always faced with choices about whether to build vs buy technologies. Our core system integrates a variety of commercial external components such as SpeedTree, DIGuy, and more, but also have many open-source components to our system such as Python (scripting), Boost (algorithms, threads, networking), and Bullet (physics). We rely on a rich data import/export toolchain via OpenSceneGraph, however, we expressly do not use any of its rendering capabilities. Having worked with OpenSceneGraph since its inception in 2000, we've found that it's very good for data reading/writing and geometry manipulation, but it's rendering is designed for GPUs from a generation ago. This meant we had to take another direction for our graphics engine and so we focused on pure OpenGL 3.3 rendering. We chose OpenGL on over other graphics technology for several reasons. First, OpenGL works on all platforms, from handhelds to desktops, independently of OS. Second, OpenGL has a rich extension mechanism, allowing vendors to expose unique hardware-specific capabilities easily. We use this to gain access to useful vendor features for advanced capabilities such as shader-controlled multisampling. Third, OpenGL is very close to the metal, allowing us to get as close to the absolute maximum performance as possible on a given GPU.

The Image Generator

In conjunction with development of our simulation engine core, we set about building an application on top of it, an Image Generator (IG) for driving simulation we call PixelTransit. Our focus was to keep the application as simple as possible, and write it the way a customer would write their own application to our engine, Blue Sprocket. This approach allowed us to accomplish the key goal we needed

[7] OS X lags OpenGL versions at this point. We handle compatibility through OpenGL extensions.

for one of our most challenging customers - build a new high-performance IG platform on a relatively new commodity graphics hardware GPU. We needed to be able to build the application logic once, but be able to change the graphics rendering underneath as the performance characteristics and quirks of the platform guided us in certain directions and away from others.

This architecture of our Blue Sprocket Engine turned out to be crucial while building the PixelTransit IG. Many times during the process of building the IG, we discovered performance bottlenecks in specific stages of both deferred rendering engine. These were either algorithmic or hardware, but in either case, we needed to rapidly iterate our design. We were able to work around particular problems within several graphics rendering pipelines by applying different deferred rendering techniques, but able to keep the core IG application structure the same.

Our Image Generation platform is compatible with either commodity synchronization solution from either NVIDIA or ATI. These allow frame frame-accurate double-buffering of graphics and GenLock within a frame. For the most part, commodity hardware is a very good choice for a modern platform, however, there are definitely tradeoffs as vendors have moved from a deep integration of hardware, including GPU, to a more integrator/assembly process. We discovered timing quirks due to various OS and hardware interactions on various platforms, necessitating rework several times as platform specifics changed slightly. COTS hardware is very wallet-friendly, but the tradeoff between up-front costs tends to get paid back in software-development time, especially as very timing-critical and bandwidth-stressing operations occur in an application.

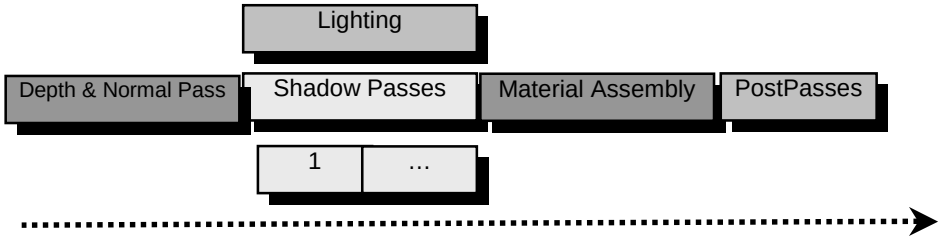
The IG networking is built off common components. We use the highly-threaded Boost library to handle asynchronous unicast and multicast network data processing. Boost allows rapid development with a modern, fully C++, peer-reviewed, and open networking stack. We created our own thin packet format for packaging up data and sending it to/from the IGs. One design decision which we made early in the system was that we needed to be able to create data packets and send them from a variety of sources. We wanted to be able to write simulations in Python or in C++ when necessary, so all our data packets have interfaces in both. We begin by writing APIs in C++ and wrapping them in Python. Our system test infrastructure is built in Python atop these wrapped APIs, allowing us to rapidly both test and develop custom capabilities.

The Graphics Engine

Early in our development process we thought we'd build our graphics capabilities on standard open-source APIs with which we had extensive experience. As we worked further through our process, various limitations became not just performance-limiting to our end goals, but completely blocked our progress with our primary goal of creating a fully deferred, unlimited-light capable image generator. Shortly thereafter we started development on the current shipping Graphics Engine for the Blue Sprocket Engine. The technologies behind this engine will be detailed next along with a coarse overview of deferred rendering.

The goal of our system was to be able to render hundreds of real in-scene lights at a time. We wanted our launch partner to be able to see actual live lights for all vehicles on-scene, and see dramatically improved lighting in urban settings at night. We quickly narrowed in on the class of techniques known as deferred rendering. Deferred rendering is a style of rendering in which you defer as much of the most expensive rendering workload as long as possible. In many simulations, this expensive work is typically the work that goes into shading pixels and fragments. In a pathologically good case, deferred rendering can mean that you only ever perform lighting and advanced material rendering once per-visible pixel (or per-fragment, if multisampling.) We chose to adapt some of the common deferred techniques to do full floating-point math, end-to-end in our pipeline, then applying a variety of post-processing effects to those floating-point values to re-range them to displayable ranges, and apply effects such as bloom, blur, and depth-of-field.

A user, in this case our graphics team, decides on an algorithm with which they'd like to render results. In our case, this was a floating-point deferred pipeline with post-processing. A user breaks down the tasks into a variety of rendering Passes and combines these through wiring texture outputs to inputs for subsequent stages. We'll presume for the next paragraphs that this pipeline has been assembled and walk through how user data flows through it. A simplified version of the deferred pipeline is represented below.



Our engine begins by taking the user Graphics Components and culling them for visibility. We developed a standalone threaded culling infrastructure so we would have flexibility in how objects were culled. Users directly insert objects to be rendered into Cull Graphs which are then responsible for computing a visible set of results each frame. Cullers are threaded using platform-native threads. Once a culler produces results, we then pass it along to a second set of threaded sort and optimize tasks which order results for rendering. On modern hardware it is particularly important to ensure that like objects are rendered together, to reduce expensive state changes in the hardware. Further, in a modern rendering pipeline, objects contributing to a scene are typically rendered twice or more depending on how many shadow maps, reflection maps, and depth passes are contributing to the scene. For these reasons, culling, sorting, and optimizing those results for display is very important to do effectively and efficiently.

After we've computed results from a variety of cullers, we pass those results to various rendering Passes which produce one or several textures as output. This chain of rendering passes together is known as a Renderer. A Renderer embodies a particular technique for rendering such as deferred rendering, forward rendering, light-pre-pass rendering, etc. Renderers can be relatively

easily interchanged, and user code remains effectively unchanged, with only minor tweaks to end-user materials to take advantage of specific Renderer features.

Renderer passes may be ordered or may be independent, allowing for coarse-threading by combining results on multiple graphics cards. This latter goal, however, remains a difficult task to do well. Core problems in this space remain with low-latency data retrieval and resubmission across graphics hardware. We are hopeful that future generations of graphics cards and APIs will allow better memory and framebuffer access across multiple cards. In the specific case of a deferred rendering pipeline, one could compute the transparent and opaque passes in parallel on separate cards prior to submission of both to the Material Assembly pass.

Finally, after a Renderer has completed it's overall scene work, a series of post-processing passes may be performed. In our deferred renderer, for example, we've chosen to compute screen-space fog, water, and snow effects based on depth results, resulting in a very inexpensive fragment operation as these operate only once per-pixel.

Results

Our overall architecture provided benefits almost immediately in that we were able to rapidly iterate our designs. However, for our end-customers are results are significantly more valuable. We'll focus on two aspects which allow our customers to derive immediate benefit, in terms of scalability, performance, and capability. These are rapid development of unique features based on our pipeline architecture, and second, dynamic performance tuning based on this same pipeline architecture.

In 2007 we developed capability for true light-lobe rendering for a client. This allowed the client to project arbitrarily-shaped spot lights into a scene and use those in actual experiments surrounding headlight design. Our deferred renderer extends this work to allow effectively unlimited numbers of spot and point lights in a scene. Each spot light can have an arbitrary shape, which means that customers can have hundreds of true-light-lobes active in a scene at any given moment. Each light interacts with the world every frame. That means that not just static objects are lit, but dynamic objects as well.

A related benefit to our customers is the ability to scale performance and quality. In deferred rendering, the performance of the lighting solution directly corresponds to the number of pixels in the final image that are lit. This gives direct control to customers to decide how much lighting realism they desire in any scene, and scale their performance proportionally. So as future hardware with more shader performance arrives, customers can directly create more lights and see improved quality, or simply keep the same number of lights and have performance improve. We strive wherever possible to expose this performance/quality control directly.

Shadows also benefit from this performance/quality control. Our engine was built to allow rapid assembly of multiple rendering passes into an overall

rendering pipeline. In our deferred pipeline, for example, users can have between 1 and 4 shadow maps computed in a directional light shadow pass. This means that as users need more quality, they can add extra shadow maps. However, we've extended this baseline scalability with further controls for users. Shadow resolution can be scaled up or down to improve quality or performance. Further, the number of samples used in filtering the shadows can be increased or decreased creating softer or harder shadows respectively. With creative use of vendor-specific multi-tap sample gathering we can create soft shadows with kernels from 9-25 taps per-result-pixel on screen.

Our engine exposes a few more of these knobs today (post-processing quality, etc) and will expose more in the future. In general we believe this is a very important way to allow customers to 'self-upgrade' as they have requirements or experiments to run which require either more performance or more realism. We believe that the graphics simulation world is only getting more complex and realistic, and customers will require ever-more control to fine-tune their simulation for their specific simulation, hardware, and experiment needs.

Conclusion

In this paper we've described our ground-up development of the PixelTransit IG and the Blue Sprocket Engine for simulation development. Our goal was to design a system that would be easy to maintain by insulating the details of modern graphics from the development of the IG features a modern simulator requires. We have an implementation which is high-dynamic range, fully deferred rendering, unlimited lights, yet with transparency, and anti-aliasing – two difficult tasks for deferred rendering. We approached the task by a rigorous study of the best practices from the games industry combined with the exacting requirements for controllability and quality from the simulation world. We did this by creating a layered architecture which allows rapid development of simulator-specific logic and features, but still allowing efficient evolution of underlying graphics rendering techniques.

Appendix: Reference Images

The images in this section show a variety of effects possible within our flexible pipeline. We demonstrate the dramatic differences in looks for a single car at different times of day, with detailed light modeling. We also show images demonstrating many real-time in-scene lights. Our poster will show more image data, results, detailed antialiasing steps, and live results.