An Internet-Based Distributed Rendering System

Ray Gardener, Daylon Graphics Ltd. (rayg@daylongraphics.com)

November 1, 2004

Updated February 6, 2006: pixel subtasking (load balancing)

Introduction

Using the Internet for distributed graphics rendering (a grid computing problem subdomain) has long been a dream of many image and animation producers. The immediate simplistic calculation suggests that the sheer number of available clients would let producers render nearly any job in very short time, or highly desirable effects such as radiosity could be furnished quickly enough for regular deployment. However, the goal has not been realized for a variety of practical factors, which I hope to address here. If nothing else, this paper should serve as a good description of the problem and possible solutions.

Grid computing in general is gaining as a viable computing model as limits to Moore's Law are encountered. It is also possible to simultaneously (and probably easily) leverage individual PCs that use (or will soon use) hyperthreading, multicore, and/or multiprocessor technology.

No particular rendering software is suggested, although many of the presently available ones could (more or less) easily be adapted for such use. Some features of what the software should support, however, emerge as a natural consequence of the problem domain.

Some Terminology

Before proceeding, a brief explanation of some of the terms used herein.

Producer	Someone who wants to produce an entire rendering. Producers use the system to enlist clients, assign them jobs, and then collect and assemble the rendered tiles.
Job	A render job for a still image or a single frame of an animation.
Tile	A rendered part of a job, usually a contiguous rectangle, scanline, column, or pixel for raytracers. A scanline renderer may conceptualize tiles as a set of one or more projected triangles to be scan-converted, which could be anywhere on the frame but are correctly depth sorted during aggregation). A REYES renderer would treat tiles as device-level geometry dicing buckets, which are similar to square raytracing tiles.
Client	A PC on the Internet configured to produce tiles. A *midlevel client* breaks up his particular tile job amongst subclients.
Dataset	A library of object models, such as meshes, along with textures.
Scene	A description of which objects are used and where and how they appear. A scene file usually contains mostly references to dataset objects.

Tree	The collective relationship of the clients to each other. The job is issued at the root node and clients at the leaves perform actual rendering. The more nodes a level has, the greater its degree of parallelism. The deeper the tree, the more serial it becomes.

The simple three-level tree that splits the job into four equal parts per level resembles this diagram:

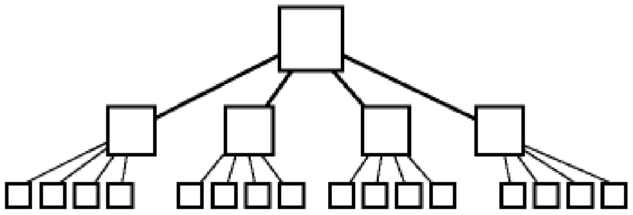

Figure 1 - Simple render tree

This diagram illustrates how such a job split might be parcelled out visually:

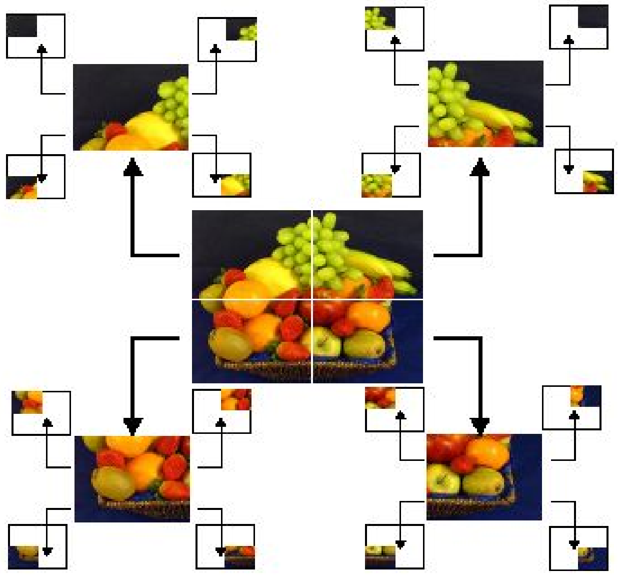

Figure 2 - Job parceling example

A render client schematically looks like this, taking in scene descriptions and using its preloaded dataset and renderer to emit tiles:

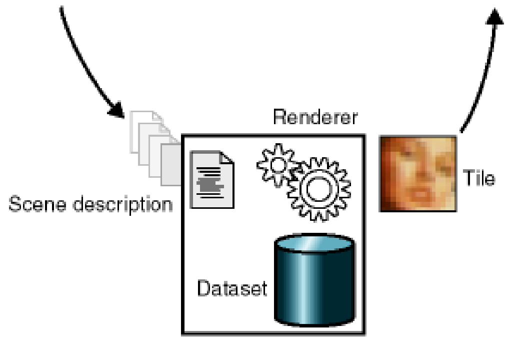

Figure 3 - render client overview

Obstacles

There are several obstacles to using the Internet (a wide area, generally lower-bandwidth public network) for distributed rendering. Some of these are common to grid computing in general.

- **Security**: clients may deliberately sabotage render jobs by returning nothing, by deliberately returning erroneous results, or by taking longer than normal. Some producers may not want clients to know what is being rendered. Producers may also try to harm clients, or third parties may impersonate producers and try to widely distribute malware.
- **Quality**: clients running across heterogenous operating environments and different ports of the render client may return job tiles which do not stitch together seamlessly, or have other artifacts.
- **Bandwidth**: Many production jobs use large datasets which take too much time to transmit to each client. Conversely, enlisting too many clients may overwhelm the producer's ability to manage aggregation as tiles are completed.
- **Storage**: Some jobs use scene description files that are too cumbersome to store on typical client hardware. Even if the datasets could fit on each client, the scene file itself (which manages the placement of objects and

refers to their shape data in the dataset) may be too large.

- **Load Balancing**: The speed of any job is constrained by the tile that takes the longest time to be returned. Delays can occur from any combination of network latency, client horsepower, and relative computational demands of the tile itself. A proper system needs to efficiently detect and break apart complex tiles and/or move them onto faster clients.
- **Compensation**: Costs demanded by clients may be collectively too large for producers to bear, or there is no practical way to spread reasonable costs among the involved clients.
- **Commitment**: The clients generally have no real commitment to the producer; the concept of deadlines and priorities may receive lip service at best. A client PC could be turned off, disconnected, or usurped for another task right in mid-render. The Internet itself has no absolute guarantee of data delivery either, and latency is both high and common.
- **Necessity**: The performance of desktop PCs makes many smaller jobs renderable locally, while commercial-grade motion picture work will continue to demand private renderfarms for the forseeable future. Does that leave enough job types in between?
- **Ease of use**: Can a wide-area renderfarm be used easily enough? If problems arise, can they be worked around in a sufficiently straightforward manner?

Necessity

I'll start with this obstacle since nothing else matters if Internet-based rendering is merely a solution in search of a problem.

Individual PC performance is good and still improving. Compared to the first PCs enlisted for raytracing, performance is spectacular. Scanline rendering has seen personal hardware-assisted solutions that are even more breathtaking. Depending on an individual's level of patience, quite a bit can be accomplished on a single computer. Since rendering is amenable to parallel computing, future CPU designs such as multicore processing will boost job turnaround times noticeably further still.

The only real practical limit to single PC performance is how efficiently electrical power can be used, since the average person will not want to significantly raise their electricity bills.

Because it offers intuitive and maximally accurate global illumination models, and as these models are computationally very expensive, raytracing offers the most reason to use Internet-based rendering. Instead of accepting ambient lighting or toned-down radiosity approximations, producers can opt to use raytracing to its fullest extent, producing images of unprecedented realism. The pyschology of raytracing should shift to using all that it has to offer, such as radiosity, subsurface scattering, photon mapping, caustics, volumetric fog and other high-quality atmospheric effects, procedurally defined shapes such as isosurfaces, etc. These features are not limited to raytracing but tend to work well with it.

Single-PC users, however, have been accustomed for so long to compromising on these effects that they have more or less come to live without them, or to treat their use as somehow “special”. It is also normal for people to be modest and conserve resources; the thought of a single person enlisting numerous (and geographically distant) clients embarasses some. While such use for frivilous jobs would indeed be wasteful, there still exist useful jobs that would benefit and should enjoy the computing power that -- however providentially -- a large number of connected computers offer.

Business presentations, a large audience, do not require great realism (in fact, this would be a hindrance since the point of presentations is to abstract away details to clarify higher-level messages). At the other end, artists seeking maximal outlets for creativity would find themselves with a fantastic tool.

To summarize, Internet-based rendering should not justify itself by doubling the speed of existing single-PC jobs (although that in itself might be noteworthy). Instead, it should be targeted as enabling jobs that would simply be impossible to do any other way. Part of the problem is that, by not having had easy access to such power, we find ourselves at a loss to imagine what we would do with it. But that should not mean that no uses exist. Even something as mundane as zooming into hithero undreamed-of depths into a Mandelbrot set image may be useful to someone.

Users should contemplate being able to produce images that have every conceivable quality option enabled, where the job turnaround difference scales up to several orders of magnitude. No one is impressed by rendering an ordinary job in two minutes instead of four, but rendering a six-hour job in ten minutes, or a two-week job in an hour, enables great possibilities. The correct audiences are those that need these vast differences in job turnaround. Ambient lighting, for example, would become an odd rarity, and radiosity would simply be “the default setting.” Producers rendering, e.g., swimming pools would think nothing of enabling caustics, and scenes without them would leave people scratching their heads as to why they were ommitted. Such producers might be jokingly told “The Internet is right there -- why didn't you use it?”

Secondary vertical-market datasets offer interesting possibilities that may be more easily justified. If clients were to preload all the terrain geometry of the entire Earth along with appropriate texture imagery and human infrastructure, producers could efficiently render detailed views and movies of any location. Potential customers include the travel, tourism, advertising, news, real estate, military, weather, landscape artistry, urban planning and civic administration industries. The actual storage required for this has been estimated to lie in the terabyte range, but hard drives of this capacity are expected to be common by 2010. For an interesting preview, readers are encouraged to try NASA's free World Wind program or Google Earth.

Quality

Quality is arguably the easiest problem to solve. It is assumed that all normal issues regarding parallel rendering (e.g., coherency of noise patterns across tiles) are already resolved, and many such render systems already exist and are in widespread use. All functional points of the render software need to be assessed for dependencies into the host environment and resolved. For example, calls into a C runtime library to use random number functions would be directed instead to private versions which would run identically on all versions of the software. Another approach is to simply favor only a small number of host environments or even one (such as Windows).

Issuing the same job tile to multiple clients allows comparisons of different client results and an ability to identify erroneous or compromised clients. Although having multiple clients perform duplicate work seems like a great waste of computing power, the Internet has so many clients that it would be remiss not to deploy them for quality control if no other means are forthcoming.

Bandwidth

Bandwidth can be solved by having producers compromise on what items constitute a practical general-purpose dataset, and then standardize upon it. Once such a dataset is available, it need only be copied to clients once (either in advance, or when a client is first requested to perform work). Versioning and absolute backwards compatability would let subsequent jobs find clients possessing the right dataset in order to produce tiles. For the purposes of this discussion, we decline direct bandwidth-increase solutions such as Internet2, since they remain either in the future or will have limited deployment not representative of the Internet at large.

A practical dataset would satisfy the diverse needs of different producers by simply being very large (as opposed to settling for much smaller, but extreme lowest-common-denominator object descriptions, such as those used by game engines that are enlisted to render machinima). This would have been impractical a short time ago, but today, typical PC storage is very large (over 10 or even over 100 GB), and distribution of large datasets is easier (BitTorrent, CD-ROM, DVD-ROM, generally increasing deployment of broadband, etc.). Since the dataset is a widely shared and collectively produced work, clients need not bother backing it up. In some sense, the popular POV-Ray renderer is a dataset, since it includes a large number of built-in shapes and textures.

The dataset conveniently solves a problem of great vexation for producers, particularly newcomers or amateurs: having a large library of premade, high-quality objects, textures, and shaders that they can just use instead of having to create. In addition to solving the problem of distributed rendering, the system also solves the problem of attracting people to make use of it.

Exactly how large the dataset needs to be, what it would specifically contain and how long it would take interested artists to develop is a subject I hope to explore soon. Even the most optimistic guess suggests an enormous deal of work. Not necessarily due to a large amount of content, but because this content must be of both uniform and high quality. If the quality is too low, not enough people will be interested in using the system, and then not enough clients will be enlistable to make the system function.

Scene descriptions are another matter. Since these vary on a per-job basis, every effort must be made to keep these as compact as possible as they will be transmitted to every enlisted client and the copying time must be included in the total production time for the job. If the copying time, for example, is equal to the average tile render time, then the job has taken almost twice as long as it ideally would have.

Having a programming or macro language in the scene description syntax (e.g., like POV-Ray) is likely vital, as repetitive constructs can be efficiently compressed. Very common expressions and functions might be candidates for placement into the dataset.

Some amount of compromise on the part of producers is necessary. The dataset cannot have every conceivable object or texture one would like, and not every scene description can be large (e.g., one controlling the position and motion of a vast cloud of individual particles). Within these constraints, however, a large number of practical jobs can be successfully rendered. If one looks at many movies, it is easy to identify numerous common objects, such as roads, trees, furniture, people, animals, liquids, clouds, rain, snow, tools, doors, buildings, stairs, vehicles, gadgets, weapons, etc. Many textures and shaders are also commonly used and can be standardized.

Overenlistment depends on how many clients can be managed before the producer's computer (or local computer network) overloads or has insufficient bandwidth to handle the requisite communications traffic. If a 640 x 480 image were issued at one pixel per client, the producer would eventually get back 307,200 tile completions. Even if only one second total is spent talking per client, that's over 85 collective hours.

A tiered system can insert mid-level managers to alleviate congestion; these units would collect finished tiles into larger ones and then pass only those onto the producer. Since the divide-and-conquer strategy grows in power exponentially, (for our pixel-per-client example) 5,120 first-level units could aggregate 60 clients each, then 85 second-level units could aggregate those, so a relatively small extra of 5,205 "middlemen" clients are needed. This works out to 1.4% of the client population, and assuming that each machine communication takes one second, the extra job time added works out to only a few minutes.

Storage

Not much can be done for jobs that are simply too large for their scene descriptions to be stored on, let alone be transmitted to, clients. Jobs of such complexity will be handled on private computing clusters or networks until more enabling technology arrives.

Again, I believe there is a sufficiently large number of more modest jobs that make a contemporary Internet-based distributed renderer feasible.

Load Balancing

Network delays (average slowness or abnormal latency) for a particular client can be mitigated by early detection and automatic enlisting of more clients to compete with the slow client. Actually, this strategy helps conquer the other related issues, client power and tile complexity. Fortunately, client power can be predicted in advance using a capabilities-exposing protocol, and tile complexity can be somewhat estimated by the producer in advance (a scanline renderer has a somewhat easier job of this than a raytracer). If a tile's complexity is badly misjudged and is larger than one pixel, we could have the client and producer automatically negotiate to subdivide the tile further amongst more clients. A recursive dynamic-enlistment scheme could even be employed starting with the producer's own PC as the first client.

Load balancing in terms of network problems is also addressed by grid computing.

Although the pixel is often considered the smallest possible tile, with some additional programming very complex jobs where even one pixel takes a single PC a long time to render could be broken up further into *pixel subtasks*. Typical motion picture frames have several million pixels, and even if only that many PCs are available, not all pixels are created equal so enlisting several to render each "hard pixel" would improve load balancing (while single PCs handle several of the easy pixels each).

Pixel subtasking involves identifying computations in the fragment rendering pipeline that can be done in parallel. An easy example is raytracer antialiasing, where multiple eye rays are fired for each pixel and averaged together.

Security

Security likely requires a multidisciplinary approach. Duplicate tile renders can be used to identify and pass over randomly hostile clients.

Unless all the hostile clients in the same job enlistment are somehow sharing information, their incorrect results will differ from each other, and it will be easy to tell them apart from the set of matching (and valid) results. Attackers could easily prearrange to just return tiles (of whatever size was requested) as flat blue patches, and if they outnumbered the honest clients, a dumb majority-rules aggregator would accept the erroneous blue tiles. Ultimately, attacks require some ability by the producer to detect erroneous tiles (the larger they are, the better, say a 5 x 5 pixel minimum, or entire scanlines) and efficiently de-enlist bad clients and re-enlist good ones.

Since the job production is batch-oriented in nature (the producer sends a request and then waits for the result), there is not much a hostile client can do except delay or return incorrect artwork. The massively distributed nature of the system also makes for a dull attack experience, since successful attacks don't really affect much.

On jobs that have been partitioned to the pixel level, it would take many hostile clients successfully returning malformed jobs to create adverse noise/snow in an animation (assuming they got past the duplicate safety checking), and the producer would simply re-render. Most attackers would probably grow bored and shift their attention elsewhere. There is no monetary gain from a successful attack, which is a further disincentive (although, arguably, someone preventing Lucasfilm from rendering JarJar Binks might feel triumphant. Then again, high-end studios are likely to continue using their own local rendering systems anyway).

Producers (or trusted authorities) can slowly build and maintain whitelists of trusted clients. Encryption can be used to prevent third-party or man-in-the-middle attacks on datastreams. Fortunately, most if not all security issues can be resolved by borrowing work already done by grid computing providers.

Using an open source system is recommended to ally suspicion of the chief software component used on each client (the renderer itself). The dataset and scene files would also be open to full client scrutiny. If a true sandbox execution environment can be imposed on the renderer (e.g., no arbitrary file I/O or instruction execution or memory addressing allowed), then scene files could be kept private for producers who want to keep their jobs undisclosed.

Further client protection from bad producers would likely involve jobs being routed or issued through a trusted authority, so producers would need to register with them (and provide sufficient credentials) before being allowed to enlist clients.

Compensation

Compensation depends on how many clients a producer enlists, and the average price each client demands. If every client was used when it was otherwise idle (e.g., using a scheduling manager that automatically enlisted clients entering nighttime), then a baseline price would involve the cost of electricity and (arguably insignificant) equipment depreciation. With enough clients, tiles could reach the level of individual pixels and the cost of compensation itself would defeat pricing, unless costs were simply aggregated over multiple jobs, and then after several weeks a client could ask for a fee. If we use PayPal, then each client would need a PayPal account and the producer would need some automated way of issuing the correct payments to each.

Obviously, accurate bookkeeping of each client's workload would need to be performed. One way to add accountability for participants is to use a by-invitation-only membership model; this allows one to backtrace up the invitation chain and query related participants (or simply delist them) whenever abuse occurs.

Since the system is generalized, it's possible that clients sometimes also take turns as producers. In this case, money would be replaced with “render credits”. Simply put, the more one acts as a client, the more one gets to enlist other clients when one wants to be a producer. A trusted authority would need to keep track of the credits on a central server.

Again, grid computing may provide answers, or enough clients exist who simply like to participate for free. Compensation can also be complicated depending on which rendering software is used and what its licence allows. Ideally, a purely open source system is used.

Compensation may be mandatory in order to avoid frivilous job submissions. For example, a producer editing a scene discovers that he can render a low-quality image on his own workstation or a high-quality version using the Internet, in the same amount of time. Either by accident or wilful abuse, he opts to do the latter each time he wishes to see the effects of his edits. Most of the enlisted clients, if they knew what the purpose of their work was, would prefer he hold off high-quality rendering until absolutely necessary.

Free enlisting would also invite hostile producers who create intensive nonsense jobs solely for the purpose of consuming compute cycles on a wide scale.

Commitment

It's reasonable to assume that the first producers will be those who are not distressed if jobs are not produced with utmost expediency. Producers on tighter schedules (e.g., they have scheduled dependant resources to be in place after estimating that a job should take a certain time to render) need more assurances.

The only solution I can presently think of is that the job distribution protocol should allow for quality-of-service (in terms of time) metadata tagging of trusted clients. However, I don't expect that even the best group of top-tier clients can ever match the delivery certainty of a dedicated private renderfarm, simply because some elements (the Internet itself) have unknown and uncontrollable quality. Those wanting a higher QoS should also expect to pay more; it is likely that clients who will guarantee their PC's availability will look for compensation. A studio that has its own renderfarm can, however, deploy the Internet to perform lower-priority jobs in parallel, thus maximizing their own equipment for top-tier tasks. The reverse is also true: during slow times, they can offer their renderfarm as a high-QoS client group.

Performance Estimates

I have not yet studied how well an Internet-based rendering system, if implemented as described, would perform and to what extent it would be competitive time- or dollar-wise to a traditional private system. However, the Internet Movie Project (www.imp.org), which uses POV-Ray as the rendering software, provides some statistics.

As a first step, however, we can describe the equations that would let us construct these estimates. Total computer time spent by the clients' portion of the network (without intermediate managing clients) is given by:

$$c\ (\ s + t_{\text{scene}} + i + r + t_{\text{tile}}\ )$$

where c is the number of clients. The other terms are per-client: s is the session overhead (e.g., locating an available client, instantiating the session, tile aggregation, and so forth), t_{scene} is the time to receive the scene description and tile coordinates, i is the time to load the scene into memory and to build any necessary rendertime data structures, r is the render time, and t_{tile} is the time to transmit back the rendered tile. The resulting number represents the collective computing power used by the network.

From the producer's end, this formula (again, without intermediates) determines job time:

$$\max(i) + \max(r) + c(s + t_{\text{scene}} + t_{\text{tile}})$$

where the maximum (worst-case) combined setup and render time cause a wait for the last tile to arrive, which is added to the time needed to talk to all the clients. The terms on the right half, as noted before, are amenable to parallel computation and indeed need to be farmed out when c is very large. c can be large even for smaller jobs since duplicate tiles may be needed for quality and security reasons.

t_{scene} is a critical factor in that every unnecessary byte slows down the result by the number of clients the producer must directly deal with (intermediate managers can forward scene descriptions in parallel). A server enlisting 100 managing clients to process a 100K scene file must transmit 10 MB of data, and each manager must then in turn transmit some multiple of that 100K as well. So before rendering really gets underway, in a proportionally scaled manager tree where the producer and each manager talk to the same number of subordinate units, we have

$$t_{\text{scene}} \cdot subordinates \cdot treedepth$$

as the minimum scene transmission time. A 1000K scene using 10 top-level managers and a three-deep tree on a 100 Kbps network would incur fifty minutes. Now this assumes that the root (producer's machine) is sequentially transmitting the scene to each top-level manager, but in practice the scene would be served by a dedicated Web host offering simultaneous downloading. Dividing even by ten gives us five minutes. The first client can also start rendering within

$$t_{\text{scene}} \cdot treedepth$$

time, which in this example is also five minutes. Clearly, the Internet is ill-suited to render jobs taking only several minutes, and this time frame is within reason for most users to let their own PCs do the work. The window opens as we get into ten-minute and longer jobs. Clients on dialup connections would be suitable only for the smallest scene files. Companies or institutions with large numbers of LAN nodes, having even greater broadband capacity, would find it efficient to render jobs even taking as little as a minute.

i is an interesting number in that for animations involving only a change in camera location, i can be made insignificant (or greatly lessened) after the first frame. This is essentially a matter of caching the rendertime data structures whenever opportunities to reuse them arise.

Conclusions

An Internet-based distributed rendering system is definitely technically possible (it has even been done by imp.org). Prototyping is straightforward since Internet Protocol is commonly used on LANs. The application-level protocols are comparable in complexity to other working systems.

The main enabler is the **shared dataset**. This is also the most time-consuming aspect to implement. Using versioning and strict backwards compatability, however, it can be built and deployed incrementally, so development of the whole system does not need to wait until the dataset is fully available.

Even when all is done, the system will not be for everyone as the dataset cannot satisfy all possible rendering situations. But there are doubtless large numbers of render projects (presentations, documentaries, scientific visualizations, simple commercials, traditional character-centric movies, storyboards, cartoons, etc.) that can be. It also makes possible a world in which anyone willing to produce a scene description can quickly obtain a professional still image or animation sequence for little or no cost.

References and/or Further Reading

IBM's Grid Computing Flyover: http://www-106.ibm.com/developerworks/grid/library/gr-fly.html?ca=dgr-lnxw01-obg-GridFlyover

www.imp.org, the Internet Movie Project.

burp.boinc.dk, the BURP net-distributed renderer project.

www.povray.org, the official site of the POV-Ray raytracing renderer.

objects.povworld.org, The POV-Ray Objects Collection.

learn.arc.nasa.gov/worldwind, NASA's World Wind planetary viewer program.

earth.google.com, official site of the Google Earth planetary viewer program.

Gritz, Larry: He posted an excellent list of obstacles to Internet-based rendering in Usenet. (http://groups.google.ca/groups?hl=en&lr=&threadm=8qnut4%24806%241%40unagi.exluna.com&rnum=2&prev=/groups%3Fq%3D%2522larry%2520gritz%2522%2520rendering%2520internet%26hl%3Den%26lr%3D%26sa%3DN%26tab%3Dwg)