ЯEFLƎCTIONS

By Andy Carluccio
University of Virginia Class of 2020
Department of Computer Science

Distinguished Majors Program Thesis

Table of Contents

If a tree falls down in the middle of the woods, and no one is there to light it, was there ever a tree at all?

Introduction

On the afternoon of August 20th, 2018, like so many other computer hardware enthusiasts across the world, I eagerly awaited the start of Nvidia's press conference live stream from the floors of Gamescon. The successor to the aging Pascal architecture was fiercely anticipated in the industry, where rival AMD had been unable to bring a competitive GPU to market in the high-end bracket for several years. Top-of-the-line graphics compute power was Nvidia's bar to set that summer, but the collapse of Moore's Law in the CPU space, as exhibited by the extinction of Intel's famous tick-tock release schedule, led to intense curiosity about how much additional performance Nvidia would be able to bring to the table after Pascal's unusually long lifespan.

After an uncharacteristic delay, Nvidia CEO Jensen Huang walked on stage with his trademark leather jacket and began what journalists would later describe as the company's most botched product rollout in recent memory. The new architecture, Turing, made spectacular promises particularly regarding its flagship feature, RTX real-time ray tracing. Unfortunately, from a poorly choreographed launch stream, to intentionally misleading marketing material, to a final product that had absolutely no developer support for RTX, Nvidia found itself in poor favor with major review outlets:

> "This is the kind of shit that gives PC gaming a bad name [...] That's what RTX is today -- it doesn't do anything. I can't benchmark goals. You're making it really hard to recommend this thing for the function that is right in the name of the product!" -Linus Tech Tips

While sales were strong, largely due to flagship purchase culture, it was clear that the "gimmicks" of RTX were suffering from an inability to demonstrate their value in consumer workloads, dominated by the PC gaming industry.

In contrast, my interest in RTX had nothing do to with videogames. While I was streaming Jensen's announcement of Nvidia Turing, I had a copy of Blender minimized on my desktop. That file contained a Python script I had been working on that allowed ArtNet DMX data to be passed into virtual lights inside of Blender. As professional lighting previsualization software products like Vision, Capture, and WYSIWYG were financially prohibitive to me, I was working on a plugin for Blender that would allow a lighting designer to create cues within a free and open source CAD program. Because of this coincidence in timing, from the moment RTX was announced on that summer afternoon, I was immediately interested in the impact real-time ray tracing could have on the theatrical lighting process.

Upon my return to UVA that fall, I connected with Professor Lee Kennedy, Associate Professor of Lighting Design in the Department of Drama. Over the next year and a half, in weekly meetings, we discussed the theoretical frameworks that would evolve into my thesis submission for the Department of Computer Science Distinguished Majors Program. While I had initially hoped to add RTX to my existing DMX project in Blender, due to the Blender 2.8 overhaul, even basic rendering support for RTX GPUs would not arrive in that engine until late 2019. Unreal Engine 4.22 was the first engine to provide support for RTX (the feature was released to developers in April 2019). Lee and I also surmised that a rendering engine targeted towards game development might provide useful capabilities such as multi-user support, virtual reality, and character integrations.

After several failed attempts at receiving funding from the University for the project, I saved money to purchase an RTX 2080 GPU for my desktop, which allowed me to finally compile my first “RTX ON” scene.

In addition to a changing the rendering engine, the overall aspirations of the project shifted from the development of a specific software tool towards research into the confluence of design and rendering philosophies. I am incredibly grateful to my DMP advisors for allowing me to refocus my project away from product development. Initially, I had proposed the creation of a complete lighting design application that supported the workflow Lee and I were formulating during our weekly meetings. This application would then be tested by users in standard UX evaluation studies, and the resulting data would be summarized and submitted along with the standalone program. As I began to develop this executable, I realized that my conversations with Lee were evolving from an analysis of the specific role of ray-tracing in theatrical lighting design into a mutual exploration of lighting mentality and a collective imagination of the tools a designer would use in an ideal workflow. As a result, I opted to de-emphasize the application itself and instead use the program I was creating as a prescriptive example of the opportunities that arise from real-time lighting visualization, summarized in a thesis that describes the workflows we explored.

This thesis brings together my notes from my conversations with Lee into a single narrative that explores the intersections of computer graphics rendering and the conceptual and visual processes at play in the mind of the lighting designer.

A Brief History of Realtime Rendering

To contextualize a discussion of the confluence of digital and mental models of lighting, it is important to first explore the evolution of computer graphics and the pipelines that support the rendering applications we use today. Across the board, digitization processes require us to think critically about the important qualities of the analog media we want to capture and process, and there have been several different schools of thought about recreating a "scene" via a digital process. My discussion of computer rendering techniques is framed around the lineage observed by the American consumer from the late 1970s onward[1]. While it may seem counterintuitive, the evolution of computer graphics processing technology has been primarily driven not by professional business workloads, but rather by consumer video games. Because of a simultaneous focus on pushing both graphical fidelity and real-time frame rates, gaming is a particularly intense workload for a computer to handle.

For most of the 1970s, almost all console games consisted of 2D sprite graphics. Atari departed from this trend in 1980 with titles like *Battle Zone*, a 3D tank simulator game. While collisions in *Battle Zone* gave the impression of mass, the actual graphics were simply a set of object boundary lines as suggested by the "wireframe" moniker given to the rendering process. The mid 1980s saw a rivalry in personal computing between Sinclair's ZX Spectrum and the North American Commodore 64. While sprites and wireframe games dominated the PC gaming market for the majority of the lifespan of these computers, games like *Driller,* released during the sunset days of the Spectrum, brought "solid" 3D to the PC, albeit at under one frame per second in a dot matrix. In 1987, David Braben released *Zarch* for the Acorn Archimedes PC,

[1] AdoredTV has a great sponsored recap of many of the games mentioned and rendering techniques used for them: https://www.youtube.com/watch?v=SrF4k6wJ-do

another solid rendering game, but at a higher frame rate and without a dot matrix display. *Zarch* was one of the first games to utilize real-time rasterization with textures, which is now ubiquitous across real-time rendering pipelines. Beginning with games like *Frontier: Elite II* in late 1993, and essentially from then on, advancements in computer graphics have been focused increasing the polygon count and rendering resolution while keeping the frame rate playable, as well as taking on additional product features designed to make the in-game lighting and physics appear more realistic.

While rasterization is by far the most popular rendering pipeline for real-time computer graphics, researchers have proposed alternative processes over the years. Before discussing the mechanics of rasterization, I will frame my future exploration of ray-tracing by discussing an older method known as ray-casting, which was implemented in a few game builds including *Wolfenstine 3D*. The process of ray-cast rendering is fairly intuitive. Recalling a pinpoint camera model, consider a series of rays passing from the origin through each pixel on the screen and out into world space. When these rays intersect with geometry in the scene, the corresponding pixel that the ray passed through is set to the color of the geometric texture at the intersection point.

By contrast, rasterization is a geometry-focused process at its core. 3D models in most rendering environments are represented as polygons of varying complexity. Rasterization determines where the vertex points of those world-space polygons appear in screen space, and then the associated textures are warped to fit to the new mapping. In rasterization, “rays” originate at the vertices and converge at the origin point of the pinhole camera, passing through the screen space. The advantage of rasterization is that fewer rays need to be cast. Assuming a 1920x1080 pixel screen, at 1 ray per pixel, ray-casting will need to cast 2,073,600 rays. By contrast, rasterization must only trace the number of vertices within view of the camera (and it

can even cull vertices occluded by other faces for further savings). While both rasterization and ray-casting have been used in real-time applications, it was obviously more efficient to rasterize, particularly in early compute hardware. As a result, more engines implemented rasterization, and hardware became increasingly optimized for that pipeline, particularly with the advent of parallelized rendering in discrete GPUs.

Over time, rending algorithms and rendering hardware have created a self-perpetuating feedback loop that strengthened the use of rasterization as the mainstream real-time rendering pipeline. From the early 2000s onward, advances in rendering have been largely focused on cramming more polygons into the screenspace, increasing the resolution of that screenspace (both in terms of viewport and texture), and adding a seemingly endless list of "rasterization hacks" that are designed to make the physics of the virtual world more convincing. In the gaming community, many have argued that the 2007 release of *Crysis* marked the last major advance in real-time computer graphics quality from the perspective of the end user. The advertising surrounding consumer graphics cards that launched in the years that followed *Crysis* began to focus more on pushing frame rates higher and higher, or bumping resolution, while turning on as many post processing effects as possible. The decay of Moore's law around this time also highlighted the reality that computer graphics were reaching a plateau on both hardware and software levels, and a major change in both architecture and algorithm would be required for the next major advance in real-time rendering.

Digital Models of Lighting

It is worth noting that neither rasterization nor ray-casting has any affordances for lighting in their core rendering pipelines; they both rely on additional processing to achieve simulated lighting effects. Rasterization in particular has no operating concept of lighting whatsoever; what it gains in efficiency over ray-casting it loses in ray data on faces. As a result, light "baking" is a popular method of creating convincing shading in real-time renders. Light interactions with the scene are calculated offline, and the resulting shadows and highlights are merged into the stored texture files themselves, which are then warped into screenspace in the rasterization pipeline. Reflections are usually simulated in a very similar way; a copy of the sky texture may be merged into a puddle texture, for example.

This process of "shading" the geometry in the rasterized scene received a formal definition in 1975 from Bui Tuong Phong, who published it in his Ph.D. dissertation at the University of Utah. The Phong model[2] of lighting considers three components: ambient light, diffuse light, and specular light. Ambient light is the baseline lighting that is not considered to be associated with any particular light source, but rather as the total omnidirectional distribution of lighting in the scene. Diffuse light varies with the geometry of the surface, using a Lambertian model. Specular light is a set of the most narrow highlights. The resulting shade of an object is the combination of these components weighted by the "shininess" of the object and related based on the relative placements and intensities of lights in the scene:

$$I_\mathrm{p} = k_\mathrm{a} i_\mathrm{a} + \sum_{m \in \text{ lights}} \left(k_\mathrm{d} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_\mathrm{s} (\hat{R}_m \cdot \hat{V})^\alpha i_{m,\mathrm{s}} \right).$$

[2] In the Phong Model above, the illumination of a point I_p is a sum where k_s, k_d, and k_a, are the reflection constants for specular, diffuse, and ambient light, α is the "shininess" constant, and the vectors L, N, R, and V relate the directions of light with respect to the viewer and under reflection.

The Phong model and its subsequent optimizations have seen widespread adoption in rasterization pipelines because it relates geometry and spatial arrangement with the shading of textures in a scene. As a result, it can be computed very efficiently in many applications. Once again, geometry presupposes the lighting under this model of rendering. In a gaming context, the performance benefit achieved from these efficient rendering and shading algorithms will be partially spent on additional post-processing effects designed to make scenes appear more realistic.

For a model that focuses on the quality of light itself first and foremost, I will build from a comparative foundation of ray-casting versus rasterization. Under the vanilla ray-casting model, a single trace is calculated to intersection points with models in the scene. A more complex variation of ray-casting, known as ray-tracing, casts up to three sub-rays from each intersection point, one for shadows, one for reflections, and one for refractions. These sub-rays recursively bounce towards the lighting sources in the scene. Each pixel is then set to a weighted average of the sub-ray intersection data. While similar to Phong in its parameterization of lighting, ray-tracing is focused on the journey of a ray, not on the specifics of the geometry in the scene. One advantage of this model is that it may consider geometry that is not strictly within the screen space; a ray could bounce out of view of the camera and still contribute to the final image. For example, a ray that bounces off of a mirror could collide with a chair located behind the camera, and the chair would appear in the final render without need for baking it into the texture. Unfortunately, ray tracing takes the already intense computational load of ray-casting and adds recursion to each ray, creating an enormous set of calculations for the computer to crunch, and as a result, ray-tracing has historically been reserved for offline computation

since its introduction to the world by Bell Labs in 1979 (the Compleat Angler took two weeks to render!).

Beyond ray-tracing is full scene path-tracing, where rays are not forced to march towards the lights in the scene but are instead permitted to freely bounce between objects in the scene for a specified period before heading towards the fixtures. The resulting accumulation of data over bounces provides highly detailed data for each pixel, but the computational expense is immense. Current efforts in real-time path-tracing leverage artificial intelligence pipelines in addition to top-of-the-line GPU hardware, and many rely on frame-to-frame inferences to improve performance.

Nvidia's Turing architecture promised to bring ray tracing into mainstream rendering, and while their RTX graphics cards were technically successful at this mission, the work going on under the hood is more complex than a straightforward implementation of the ray-tracing algorithm. While there are dedicated "RT" cores on Turing which are optimally designed to trace rays, they are only capable of casting one to two rays per pixel at standard HD resolutions. Under most circumstances, the resulting image would be so noisy that the renders would be essentially useless, even if they could be produced in real-time. To combat this, RTX was implemented as a hybrid renderer. RTX GPUs only calculate ray traces for objects in the screenspace that are designated as meeting certain criteria that would result in a noticeable difference in presentation if the surface was ray-traced versus rasterized. For the surfaces that are designated for ray-tracing, artificial intelligence running on the GPU Tensor Cores plays a critical role in denoising the ray-traced renders, using a variety of runtime denoising filters to create surprisingly similar images to those that are fully ray-traced, while still operating in real-

time. The RTX hybrid render brings the graphical advantages of ray tracing to the table without the crippling computational burden. Well-optimized RTX implementations in modern AAA games see only a 40-60% reduction[3] in frame rate with RTX enabled over fully rasterized scenes. Nvidia's 2nd generation Deep-Learned Super Sampling technology (DLSS-2) has also helped improve performance, allowing the scene to be rendered at a lower resolution and then intelligently up-scaled to the native resolution. By leveraging AI capabilities on the Tensor cores, faster tracing due to dedicated RT hardware, and intelligent selection of surfaces to ray-trace, RTX has brought real-time ray-tracing to the consumer graphics market.

[3] Technology news outlet GamersNexus maintains a large database of well-tested game performance benchmarks: https://www.gamersnexus.net/features/game-bench

Ray Tracing and the Lighting Designer

Having discussed the technical foundations of rendering with respect to lighting, I now relate these models to live entertainment lighting design. My weekly conversations with Professor Lee Kennedy from the UVA Drama Department over the past year and a half were critical to my formation of a mental process for lighting design. I found that these techniques for designing and implementing theatrical lighting frequently intersected with the different computational lighting models previously discussed.

Theatrical design, across disciplines, is focused principally on the perspective of the audience. At least in a traditional context, live entertainment is an intentional, curated experience for the spectator; extensive considerations are not required for spatial points of view that a patron will never occupy. At the same time, theater facilities come in many different configurations, and within those spaces, there are almost always variations of viewpoint depending on where a particular patron is seated. Scenic, audio, and lighting design must consider these variations, while simultaneously squeezing out as much efficiency from their designs as possible to save budget.

Lighting holds a special place in entertainment visual design; it in essence holds the "final say" of what a patron does and does not see. Additionally, and perhaps most critically, lighting determines *how* the audience perceives what is on the stage. For all the work that will go into designing the costumes and scenic units, these elements will be practically invisible if the lighting designer does not illuminate them. Beyond revealing actors and objects on stage, light also can use shape and color to provide variation to the static tones and pigments of actors and scenery.

Lee and I have regularly discussed and debated whether lighting is itself an "object oriented" phenomenon. Ultimately, it is a matter of design perspective on what the "object" in a lighting context is. An intuitive object for light is the fixture emitting the light, which can be concretely described, and often serves as the basis for lighting parameterization. Luminosity, lens data, pan, tilt, zoom, color, and gobo are just a few discrete quantities provided by a mounted fixture that can be used to describe lighting, and under this model, the designer must also act as a technician of sorts to determine how the control of these parameters culminates in a desired aesthetic. I refer to this as the "forward direction" of lighting design conceptualization.

An alternative model for the lighting "object" is the set of surfaces it interacts with on stage. If the primary concern of the designer is the perception of light itself, separate from or in addition to the objects being lit, it naturally follows that the final light "bounce" from the stage to the eye of the beholder is where design attention should be primarily focused. The optical properties of the surfaces in the scene play a critical role in the quality of this final bounce, and these surface parameters themselves are quantifiable. Transparency, reflectivity, refractivity, color, and relative placement are a few parameters of surfaces that influence their perception under light. With this object definition, the designer operates as a "light painter," deciding how the audience member should view the stage by applying and shaping illumination through parameterizing the final bounce. I refer to this as the "backwards direction" of lighting design conceptualization.

In practice, lighting designers are likely to leverage both forwards and backwards mental models to suit their own problem-solving needs. Presently, it may seem absurd for a lighting designer to completely ignore either their fixtures or their surfaces, as both are profoundly influential on the final product. Flexibility to operate in both directions is certainly important, but

during the design workflow, the software tools currently available to lighting designers are incredibly skewed towards the forward model, creating a focus on the lighting fixtures themselves. This bias crops-up during conceptualization; visualizations from the simulated patron viewpoint are mostly a means of collecting feedback to potentially iterate on fixture layout. Instead of creating designs within the visualizer, the user generally ends up only testing and tweaking their existing designs. The root of this phenomenon in lighting previsualization is inherently related to the difference between rasterization and ray-traced rendering pipelines.

The most popular entertainment previsualization programs on the market (Vision, WYSIWYG, and Capture), all use rasterization for their real-time viewports. All three applications encourage the user to import or generate geometry, hang virtual fixtures, focus those fixtures, and repeatedly iterate until a satisfactory design is reached by using visual feedback from the rendering engine to anticipate the consequences of making different parameter choices. Lighting visualizers are mostly used to generate design renderings (often in conjunction with Photoshop for additional compositing flexibility) as well as to program cues into a paired lighting control console (streaming DMX control data between the console and the visualizer to simulate what the console configuration would do to the real scene without being physically connected to the venue's lighting system). Obviously, the ability to iteratively design without a physical link to the venue is a collaborative and logistic necessity, and these applications have seen widespread adoption as the digital alternative to purely pencil-and-paper plots.

Due to their architectural emphasis on the forward model, current lighting previsualization software solutions tend to fill the role of engineering aids rather than design assistance programs. While they are excellent at generating electrical paperwork, setting focus points, and building cues outside of a performance venue, these programs paint in broad strokes

when it comes to the subtly of design processes. Despite these applications having multiple pages of rasterization settings, allowing the user to specify which “hacks” to use to approximate what lighting would look like in the actual venue, their usefulness in the visual design workflow itself is quite limited. At least in WYSIWYG, which began essentially as a wireframe beam-angle calculator over twenty five years ago, the main features that add to “realism” have been the result of substantial rasterization post-processing development efforts in their custom rendering engine over the years. For example, even fundamental capabilities like real-time reflections (mirrors) and alpha beam shadows (light shining through a tinted surface), have only been made available in the past few years, and at significant runtime performance expense in many cases. The rasterization pipeline underneath all major lighting pre-visualization software forces the designer to choose between graphical fidelity and real-time performance, both of which are essential for iterative design processes. The development release notes of major visualization applications over the past two decades chronical a struggle against the limitations of rasterization, where performance gains from Moore’s Law compute increases are traded in for additional simulation options in the settings menus.

As I have previously discussed, rasterization is a projective process. It is focused on the mapping of vertex locations from world space to screen space, and it uses that mapping to warp texture data to appear properly on screen. Shading is a secondary process used to modify texture data, leveraging information about the lighting fixtures and a small set of material interaction definitions to simulate light within the scene. Ray-tracing flips the rendering pipeline into the reverse direction, focusing on the paths from the screenspace to points in the worldspace. Rays recursively march towards the fixtures, but are weighted to emphasize interactions early in the chain from the perspective of the camera, leveraging a fixed set of sub-rays in reflection,

refraction, and shadow to shade the texture files of the intersected geometry. Rasterization is the natural rendering method for forward lighting conceptualization, and ray-tracing is the closest model for the backwards process that can run in real-time.

It is no surprise that visualizers have relied on rasterization pipelines; ray-traced alternatives were simply too slow for real-time work. Major visualizers often do contain a ray-traced offline renderer, promoting them as the final check to render the most “realistic” scene, though users are often prone to just export the real-time viewport to save themselves from an unpleasantly long ray-traced render, particularly if the rendering from either export will be passed into Photoshop anyway for additional editing. This visualization pipeline necessitates the forward model of lighting design, encouraging designers to think like electricians from a fixture-centric position. At a computational level, rasterization is reinforcing this philosophy of lighting design in every visualization program currently on the market. To build a tool that allows for flexible design thinking in both the backwards and forwards directions, hybrid ray-tracing must be used as the underlying rendering pipeline.

Thesis Chronical

Formulating this thesis was itself an iterative design process. My initial expectation was that I would quickly enable RTX hybrid rendering in a test scene in Unreal and spend the majority of my time building an executable design program around this feature, much to the praise of lighting designers as observed in detailed user experience studies. Everyone would applaud my new program because “ray-tracing is just better.” A more realistic render would be generated in my software, and so long as my interface choices did not impede the design process too severely, I would have created the foundations of a market-leading lighting visualization product. I reflect on these naïve assumptions with abundant amusement, and I am thankful that the structure of this thesis program allowed me to move my own goal posts as needed until I could arrive at a conclusion supported by my investigations.

It turns out that the only assumption I had at the onset of the project that still stands is that adding RTX support to an Unreal project is relatively straightforward. Using UE4 4.23, any scene is only a few clicks away from rendering in DX12 with hybrid RTX rendering available if a supported GPU is detected. My first surprise in the development of my application was that few elements experienced noticeable change when switching over to RTX. This is due to the critical role that material properties play in the hybrid rendering process. To determine which surfaces should be ray-traced and which should be rasterized, the engine will use program-defined thresholds to organize materials based on a set of parameters, particularly roughness and transparency. If the material is smooth and shiny, it will be ray-traced. Otherwise, a ray-traced reflection is unlikely to be observed anyway, so the engine will rasterize the object because it is faced with the rougher materials. Ray-traced shadows are also determined by the user. If objects and lights within a scene do not move, shadows will be baked into the texture because they can

be pre-computed and then simply loaded at runtime. Dynamic shadows will use the RTX shadow sub-rays. Initially, I had not considered ray-traced global illumination to be of much importance to the lighting design workflow because I was still thinking in the forward direction at the time; I surmised that having intentional lighting placements would be sufficient for crafting the final image for the audience. Once I began to experiment with global illumination, I realized the power of ray-tracing for placing scenic elements into conversation with each other. Suddenly, the window curtain was casting red light onto the white wall next to it. Sculptures on bright table tops were illuminated from indirect light bounces from below. The scene as a whole immediately felt more unified. This was the closest I would get to RTX "just working," as for seemingly little effort, I suddenly had incredibly complex lighting effects that would require intense "rasterization hacking" to achieve in other engines.

By October 2019, I had worked out the settings I could control within the render volume, and I began to investigate the import pipeline for 3D geometry into UE4 at runtime. I quickly discovered that, while Unreal is incredibly well-optimized for the rapid development of videogames, attempting to do absolutely anything outside of the mainstream, such as summoning a File Explorer window at runtime, is enormously difficult. Thanks to a few experimental features within the engine and several sleepless development weeks, I was able to add support for the live import of .obj geometry into my application with the hybrid renderer active. Geometry segmentation was a critical component of this import; without the ability to specifically address particular faces of geometry in the scene, RTX and any CAD-oriented features would be completely useless. Correctly applying the image files described in the geometry's accompanying .mtl file also required extensive labors, though I did eventually gain the ability to import complex theater models into my UE4 project at runtime. On the tail-end of

this incredibly tedious development process, I came to appreciate the importance of material data for the hybrid rendering pipeline.

During the majority of November and December, my conversations with Lee began to shift towards conceptualizing fixtures within software. This was the critical turning point for my thesis project, where I began to realize that ray-tracing was opening the door for a different way of thinking about the design process, and that exploring the consequences of that mental model would be more valuable than implementing a production-ready executable. Our weekly discussions were rooted in questions about abstracting versus simulating light. Is it necessary for a good previsualization application to perfectly model every portion of a fixture for highly accurate simulation at runtime, or can the properties of a fixture be abstracted to a diffusion image that could be projected through by a simple point source to simulate the complex fixture? This led to an exploration of IES light profiles, which I added to my application so that I could import standard theatrical lights within the ray-traced scene.

Over winter break, I was commissioned by a private entity (separately from my thesis project) to create a computer vision algorithm that could determine the 3D location of a fixture based on an image of its footprint in realtime. After creating this algorithm, I realized that the consequences for my thesis and the lighting workflow were immense; a designer could simply "paint" light on their scene and then "compile" where the lights they owned needed to be hung in the theater in order to achieve the desired look. Due to a NDA, I am not at liberty to disclose the algorithm in this paper nor could I integrate it into my project, but I am confident that such a workflow could be established thanks to the research I completed for this separate entity.

In January, I added a critical feature to my Unreal project that reinforced my expectation that utilizing the tools available in modern game engines would allow me to deploy workflow

capabilities that lighting designers would find valuable beyond just ray-tracing. Mainstream lighting previsualization software solutions have comically poor “humanoid” models across the board. In just a week, I was able to add in highly realistic, animated human figure models to my project, complete with the ability to possess them in third-person, place them in the scene, and record their movements to cues to bind to different scene transitions. I quickly developed a workflow where I could spawn a set of characters, pilot them into position, set their idle animations, return to a first-person viewpoint, and see how they interacted with the lights I placed in the scene. This model placement procedure was infinitely more helpful than mainstream alternatives. These characters brought the scene to life in a way that stand-ins from existing applications could not approximate. While their realistic appearances were certainly helpful for understanding their interactions with the lights in the scene, the subtly of their movement animations, especially at idle, provided an astronomical amount of context to the behavior of the bodies under light. Watching these characters run around a virtual copy of UVA’s Caplin Theater under ray-traced lighting was a groundbreaking moment for the project.

Finally, in February, I added my final feature to the project, parsing ArtNet DMX data into virtual lighting features within Unreal. This allowed me to connect to a virtual console, patch into my UE4 lights, and control them like I would an actual theatrical lighting system. At this point, all of the main workflow elements were modeled within my software, but due to my breadth-versus-width development approach, where I wanted to explore as many aspects of the programs as possible without fully debugging and implementing each feature, the program is not in a state where it should be used for a professional project, but rather it serves as a reflection of the ideas and discoveries that Lee and I have explored over the past year and a half.

Concluding Thoughts

My original thesis, using real-time ray-tracing only as a solution to the problems presented by rasterization, is admittedly short-sighted. While ray-tracing can consolidate most of the post-processing effects commonly used to rasterize a lit scene, and although the RTX pipeline can create these renders at similar speeds to heavily processed rasterizations, I drew several more fundamental conclusions from my exploration of design-thinking with a software perspective. I found that the architectural differences between rasterization and ray-tracing paralleled the workflows that can be created within each pipeline, and a previsualization tool built around ray-tracing could be best seen as an opportunity to create a new devising method for computer-assisted design in lighting for live entertainment.

As a student of Computer Science in the College of Arts and Sciences, intersectional studies between technology and art have been at the heart of my undergraduate education. The benefits of applying a technological approach to artistic processes are numerous and easy to observe in my own workflow; applying the scientific method when solving design challenges has been fundamental to my participation in the arts. Particularly in my projection design work, where media server programming is critical, lessons I have learned about algorithm design and diagnostics have been incredibly valuable. While it may be less visible, I have also found that artistic devising processes revise or re-emphasize the scientific method of technical problem solving; the practice of live, transparent revision, particularly in collaborative settings, has made me a better computer programmer. Thinking about software as an artistic design problem and using the devising methodologies broadly promoted in my arts education have been key to my software development approach. I began my thesis studies focused on how technical abilities could improve the design processes of lighting designers, but I find myself at the end of this

project more interested in how existing mental models within lighting design can be technically expressed in the development of entirely new software workflows.

The application of iterative design to the software development cycle is the most important lesson I have learned from my thesis studies. Creativity is a shared foundation between arts and sciences, but I find that revision at both micro and macro levels appears to be absent in computer programming discourse. Algorithms are debated and decided upon, and then rendered into software. If revisions are needed to satisfy requirements, the cycle is repeated, usually attempting to preserve as much of the previous infrastructure as possible for frame to frame persistence. Key points from solutions move from mind-space into software-space, and the webbing between these solution vertices are subsequently derived. It is a discrete, digital procedure, at best a stochastic gradient descent until arriving at the ideal solution. On the other hand, devising in the arts is an analog workflow (to borrow signal science terminology); work exists in a process of constant discovery and revision at every stage, where ideas are freely expressed and counter-balanced throughout the lifespan of the piece, even after the designers have removed themselves physically from the process, making the very concept of a "final product" particularly nebulous in live performance settings. Implementation in performance is the physical rendering of the conceptual subjects usually created by collaborating artists reflecting, refracting, and shading ideas between each other. Lessons from these embodied experiments update the mental model in real-time, expressing constant backpropagation and recursion. Design solutions evolve into being, whereas technical procedures seek to arrive at specific conclusions. Metaphorically, designs are traced and technologies are rasterized.

The ideal tool for a lighting designer must, at an architectural level, think *with* the artist. Using a reverse model of lighting, where the rendering power is focused on what the audience

can perceive, a fundamentally new visualization program will compute the consequences of choices made by the designer, placing lights as necessary to achieve the desired “look.” It must also act as a practical check on each design choice, warning the user if the decisions made exit the realm of possibility in the context of the available lights and hanging positions. The ideal visual design software will efficiently keep pace with the user, providing the maximum amount of realism as quickly as possible. But it must also exist in a collaborative space, allowing multiple designers to view the scene together, either in virtual reality or on screen. Critically, the software needs to present living scenes to the designer, with stand-in subjects that appear and behave like real performers. The tool must allow for the designer to think both realistically and without unnecessary cycles of trial and error that can be automated away.

My thesis project transitioned from an implementation of idea into a sandbox where new ideas could be formed. By approaching the problem of building a lighting design software tool as though it were an artistic design project itself, I was able to examine the heart of the issues behind current visualizers on the market and conduct experiments that lead me to discoveries suggesting a new workflow altogether. In studying the intersections of ray-traced rendering techniques and the mental models of lighting in visual design, I found that improvements in visual fidelity were less important than the value added from careful attention to object and surface parameters, light source diffusion models, and realistic figure animations. Working in an established visual design process with a desire to make it more efficient, I made discoveries through experimentation that lead me towards a fundamentally new workflow altogether. Most importantly, I found that when the developer and the designer arrive at a mutual understanding of their creative processes, the possibilities of what they can generate together are truly endless.