An Approach to a High Fidelity Real-Time Renderer in Metal

D. James Kelly
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
dxk2294@rit.edu

Abstract—We introduce a physically based real-time renderer, Arsee, designed to take advantage of the low level platform specific features of the GPU hardware-accelerated API Metal to provide performant realistic rendering of complex scenes. Arsee is designed to be used for both real-world applications and educational purposes, allowing users to examine how the engine works in real time.

Index Terms—Computer Graphics, Real-Time Rendering, Physically Based Rendering, Swift, Metal, macOS

I. INTRODUCTION

Game engines are large software frameworks designed to simplify the process of creating video games by encouraging code reuse and reducing the time it takes to achieve a first playable prototype. These engines are composed of many subsystems that are responsible for managing one aspect of the game development process. For example, a typical modern game engine provides functionality for rendering 2D or 3D graphics, animation, sound, physics, networking, artificial intelligence, input, custom scripting, and more. These engines also provide separate UI-driven developer tools designed to work in tandem with the engine to increase developer throughput. As these engines are complicated and costly to produce, most modern games are developed using commercially available game engines that can be acquired by paying license fees or royalties.

The most commonly used game engines include the Unreal Engine and Unity. The Unreal Engine is a proprietary engine written in C++ that has been primarily targeted towards the development of first-person-shooter games. It is renowned for its graphical capabilities, though other engines in the field have begun to match its prowess. Unity is another proprietary engine written in C++ with an exposed C# API. Beginning as a product tailored towards Mac OS X, the engine has since evolved to target 27 separate platforms. Due to Unity’s low barriers to entry, including a free subscription and abundant documentation, Unity has found itself a popular engine for hobbyist and independent game developers. It is often praised for its entity-component-system (ECS) architecture, a design pattern that lends itself to more easily developing games.

In an ECS game engine, games are architected to encourage composition over inheritance. One famous problem in inheritance-based object oriented programming is known as the "Deadly Diamond of Death", a situation that arises when one class inherits features from multiple parent classes. This situation leads to ambiguity as to what implementation the subclass should use. If a game engine were to encourage inheritance-based object oriented programming, the "Deadly Diamond of Death" would occur with more and more frequency as the complexity of the game increases, as the distinctions between classes become muddled. An ECS game engine resolves this problem by flattening the inheritance hierarchy. Components define a set of related state and how they interact with other components of the same or different types. Entities, or game objects, are IDs that are associated with components. We say that entities are composed of components. In pure ECS architectures, behaviors are spun off from components into separate systems that act on collections of components. This leaves components solely as state containers. This architecture allows each component to be independently updated, and also allows for complex entities to arise out of fairly basic subcomponents.

The Unreal Engine and Unity, though popular, have been forced to become overly generalized in a bid to cater to all possible markets. The Unreal Engine has lost its focus on first-person-shooter games and has instead shifted to generic features that can be applied to most game genres. Unity has become a "jack of all trades, master of none," leading to many of its games looking and playing in a very similar fashion. When Unity shifted to becoming multi-platform, it lost its ability to exploit Mac OS X specific features. Though it features a Metal renderer for macOS and iOS builds, it does not take advantage of all of the features Metal provides to enhance performance. A niche has emerged in the market as a result of the over-generalization of modern game engines.

In this paper, we present a new game engine, Arsee, written in Swift. It is designed to exploit the language features of Swift to reduce common pitfalls in game development. It features a high fidelity 3D rendering engine written Metal that is physically based with support for many graphical features found in modern games. As the engine is specifically targeted towards macOS, we can take advantage of the platform-specific features of Metal to optimize performance in GPU intensive scenes. We have developed a simple scene to demonstrate the feature set of Arsee and compared its performance.
to a similar scene developed in the Unity engine. We present these results in section VI.

II. THEORY

In this section, we will discuss the theory behind the techniques we employ in Arsee. Real-time renderers must compromise between performance and accuracy, and it is important to understand how light travels in the real world so that we may accurately approximate it in real time.

A. Rendering Equation

The rendering equation[1] represents the complete transport of all light in an enclosed scene:

\[ L_o(x_\omega_o, \lambda t) = L_e(x_\omega_o, \lambda t) + \int_{\Omega} f_r(x_\omega_i, \omega_o, \lambda t) L_i(x_\omega_i, \lambda t) \cdot n \, d\omega_i \quad (1) \]

This equation essentially states that the lighting leaving a surface is equal to the light reflected by the surface and the light emitted by that surface. Though, conceptually, it is quite simple, solving this integral equation in real-time is currently not computationally feasible. Modern real-time renderers are the culmination of decades of heuristic techniques that, through the power of GPU-accelerated computation, can accurately approximate the rendering equation. We will discuss several of these techniques in the remainder of this section.

B. Surface Shading

In real-time renderers, surfaces can be lit in primarily two ways: direct and indirect lighting. Direct lighting occurs when the photons emitted by a light source directly strike a surface without hitting any surface along the way. That is, the surface is directly lit by the light. Indirect lighting occurs when photons bounce around the scene, striking other surfaces, absorbing and losing energy of different wavelengths before striking the surface.

For simple renderers, we tend to “hack” indirect, or global, lighting by adding a fake ambient term to every lighting calculation. This ambient term represents the light that bounces around the scene and eventually ends up back at the surface. For very simple systems, this ambient term is a small multiplier of the incoming light’s radiant flux.

Direct lighting is, computationally, quite simple to calculate. We can break up direct lighting into two components: diffuse and specular lighting. When light strikes a surface it can either reflect (specular) or be absorbed by the surface (diffuse).

The diffuse lighting component represents light that penetrates a surface, bounces around inside the material, and eventually exits the surface near where it penetrated. Because the distribution of this light is relatively uniform regardless of the viewing angle, we can easily approximate the diffuse lighting of a surface with simple Lambertian reflectance. That is, the diffuse light that leaves a surface is dependent on the angle between the incoming light vector and the surface normal. The closer the two are aligned, the more light tends to reflect off of the surface.

The specular lighting component is more complicated. Visually, specular lighting tends to appear as tight, bright, white highlights on the edges of objects. It tends to make objects appear shiny. When light strikes a surface, the distribution of purely reflected light is not uniform. Virtually all purely reflected light reflects perfectly across the normal of a surface, and therefore specular lighting is dependent on not just the surface normal and direction of light, but also the direction from which the scene is being observed. This results in the specular highlight of a surface appearing to “move” as the viewer moves. There are many different methods of calculating this surface highlight. Blinn-Phong, a classic method of surface shading, computes the halfway vector between the view direction and light direction and uses the angle between the halfway vector and the surface normal to compute specular lighting. This method, though computationally cheap, does not quite accurately model how specular lighting really works.

Many classic methods, or bidirectional distribution functions (BRDF), for computing surface shading, such as Phong, Blinn-Phong, Flat, or Gourad do not adhere to the laws of physics and energy conservation. One modern approach to shading, physically based rendering, attempts to solve that.

C. Physically Based Rendering

Much of real-time computer graphics concerns itself with surface shading. As GPUs have become more powerful, more complex methods of calculating surface shading have risen. The most common approach used in modern renderers is known as physically based rendering (PBR). In it, we attempt model objects as closely to their physical properties as possible. This allows artists to define models in terms of their physical properties, and stops artists from needing to tweak assets depending on the environment in which they will be placed.

1) Metals and Nonmetals: PBR break up objects in the real world into two types of surfaces: metals (conductors) and nonmetals (dielectrics). As light tends to interact with these two types of surfaces differently, we denote each surface as being either metal or nonmetal.

Non metallic surfaces, such as wood or plastic, exhibit both specular reflection and diffuse refraction. As described previously, when light interacts with a non metallic surface, a percentage of it will penetrate, refract, and resurface and a percentage of it will immediately reflect. For most non-metals, approximately 5% of the light reflects. One interesting characteristic for specular light for non metallic surfaces is that it is almost always grayscale. The surface is not biased in terms of what wavelengths of light it reflects. This is what leads to the white reflects we know on most surfaces.

Metallic surfaces, due to their tightly packed atomic structure, do not allow light to easily penetrate. Therefore virtually all light is reflected in a specular fashion when striking a metallic surface. There is no diffuse refraction when computing the surface lighting for a metallic surface. Unlike non-metals,
However, the specular component is often times tinted a certain color for metals. For example, gold is a metal and exhibits a gold color even though all the eye sees when viewing gold is the specular reflection of light.

2) Microfacets: If we imagine the surface of an object at a microscopic scale, we can see that even an object that appears smooth to our eyes is actually composed of a series of ridges. The rougher an object is, the steeper and more numerous these ridges become. As we saw earlier, specular reflections depend on the angle between the halfway vector and the surface normal. If we think of these ridges as perfect mirrors, then we need to be able to approximate how many of these perfect mirrors we can actually see from any given angle. If we can approximate this value, we can use the value as a multiplier in our specular lighting calculations. The rougher an object’s surface, the more spread a specular reflection is. The smoother an object’s surface is, the tighter and brighter that reflection appears to be.

3) Direct Lighting: In the previous sections we discussed metalness and roughness. If we model an object using these properties, we can apply a set of equations with these parameters as input that more realistically approximate the interaction of light with a surface. In physically based rendering, we still compute direct lighting as expected. Direct lighting is broken up into a diffuse refracted component and a specular reflection component. But now, we adhere to the physical properties of the objects we are rendering and also make sure to conserve the overall light/energy of the scene.

Diffuse lighting in a physically based system is actually identical to classical methods. As before, we calculate the diffuse lighting component as a product of the incoming radiant flux and the angle between the surface normal and incoming light direction. This time, however, we must be sure to not violate the law of energy conversation. We calculate the specular component first and then the fraction of the light that does not reflect must be refracted. We use that fraction as a multiplier of the diffuse component to calculate the true (energy conserved) diffuse refraction for the surface. If the surface is metallic, we do not calculate any diffuse lighting component, as any refracted light remains trapped within the metal.

Specular lighting is, again, far more complicated to calculate than diffuse lighting, as it is dependent on the viewing angle. If we consider our previously described microfacet model, then the surface of all objects is composed of jagged ridges that act as tiny perfect-mirrors. We need to incorporate the roughness of a surface into our lighting calculation. There are many different methods of doing so, but the bidirectional distribution function most real-time renderers use is known as the Cook-Torrance BRDF[2]. The specular component of the Cook-Torrance is denoted by the following equation:

$$k_s = \frac{DFG}{4(V \cdot N)(N \cdot L)} \quad (2)$$

where D is the normal distribution function, F is the Fresnel equation, G is the geometry function, V is the view direction, L is the incoming light direction, and N is the surface normal. There are many different ways of implementing D, F, and G, so we will just describe their purposes here.

- Normal Distribution Function - calculates the percentage microfacets that are oriented towards the viewer. The smoother a surface, the tighter and more focused the specular reflection becomes.
- Fresnel Equation - used to approximate the physical phenomenon where, the closer to 90 degrees the angle between the viewer and the surface normal becomes, the stronger the specular reflection becomes. When the angle reaches 90 degrees, surfaces exhibit total specular reflection. This equation also requires the “base reflectivity” of the surface, which determines the surface color of metals and the reflectivity of nonmetals.
- Geometry Function - calculates how much light can escape a surface after self-shadowing and self-obstruction is taken into account. The smoother a surface, the more easily light can escape as the microfacets do not trap as much light.

By combining these three functions, each of which are dependent on physical properties of the material, we can much more closely approximate the rendering equation for directly lighting as compared to previous methods.

4) Image-based Indirect Lighting: Image-based indirect lighting attempts to improve the ambient term we discussed earlier. It samples a cube map, representative of the scene, and treats it as a global/enveloping light source. Therefore, we remove the old ambient calculations and replace it with two new terms - the diffuse global irradiance and the global specular component calculated from a series of precomputed maps. Typically we take a high dynamic range image as the skybox of a scene and perform further operations on that map to produce the maps we need for image-based lighting. This way, we can give objects a sense of belonging to the scene in which they are rendered. This is a fairly detailed and complex process, and detailing it is beyond the scope of this paper.
III. IMPLEMENTATION

A. Gamma Correction

Humans perceive brightness in a non-linear fashion - we can more easily perceive minor differences between very dark scenes as opposed to minor differences between very bright scenes. Therefore, scales we perceive to be linear actually exhibit a bias towards the darker end of the brightness spectrum. Because we are used to this behavior in real life, modern monitors perform this mapping for us - this is known as gamma. Colors in linear space are automatically mapped to the gamma space of the monitor, typically by taking the linear color to the power of 2.2.

To ensure our linear-space lighting corrections appear correct, we apply gamma correction to our assets. Immediately before displaying our rendered image to the user, we essentially negate the gamma of the monitor by taking the linear color to the power of $1 / 2.2$. This is then corrected by the monitor, essentially negating our gamma correction. The result allows use to do our calculations in linear space and not have to worry about the gamma of the monitor.

Artists, who often do their work on computer monitors, are producing their assets in the gamma space of their monitor. These values, while they appear correct to the artist on their screen, are not physically correct. This provides poor results in complicated lighting equations, such as those utilized in physically based rendering. We also must convert these assets into linear space by performing inverse-gamma correction.

B. High Dynamic Range

In photography, high dynamic range refers to the process of taking multiple photos of the same scene with different exposures, and then combining those photos together. This process allows the camera to capture high-detail in scene regardless of how the lighting conditions vary throughout the image.

We can apply a similar technique to our rendering. Rather than represent our light sources a emitting colors between 0.0 and 1.0, we can represent our light sources as emitting energy of certain wavelengths (red, green, and blue). We can then model our lights using physically realistic values, as opposed to having to fit within the arbitrary boundaries of 0.0 and 1.0. We perform our lighting calculations as normal, and instead store our values in a frame buffer that can contain values greater than 1.0. From here, we can perform additional calculations on a buffer that contains physically accurate light values.

Eventually we have to map this high dynamic range back to the range 0.0 to 1.0. For this, we apply tone mapping. There are many different approaches to tone mapping, each of which favors different ranges of values. We use the Reinhard tone mapping algorithm, which attempts to evenly balance brightness values across the range 0.0 to 1.0. This algorithm is typically applied right before gamma correction.

C. Color Picking

Often times, in both games and tool development, it is useful to be able to select objects in a rendered scene. For example, scene artists would like to be able to click and drag objects throughout the environment with ease. Players may want to be able to pick up items with a simple click. A complete solution would arrange the scene into an object hierarchy with nested bounding volumes, and then fire a ray into the scene when the mouse is clicked. The ray would then have to traverse the hierarchy and ultimately do ray-triangle intersection with any mesh to determine whether or not the object was clicked on (there was a collision). This is not only an expensive operation in terms of processor time, but it also requires a lot of effort on the programmers part to set up.

An alternative method, known as color picking, attempts to get around the computational and programming expenses of ray casting. When a user clicks on an object in a given scene (Figure 2) a series of shaders are executed in the background, hidden from the user.

First, we render all visible objects with a unique color (Figure 3).

We then batch all of the user’s mouse clicks within the current frame and send them to the GPU to retrieve the colors associated with the pixel locations underneath the click locations. We send that data back to the CPU, where we can extract the colors and perform a lookup per color to determine which objects were selected. From there, we can render each selected object to a separate buffer with a specified highlight color (Figure 4).
After rendering these objects, we blur the frame buffer using a separable gaussian blur pass with ping pong buffers (Figure 5).

Finally, we can subtract the solid colored frame buffer from the blurred frame buffer, and add that result onto the initial frame buffer. This results in a cheap and easy highlighting for selected objects.

This approach, coupled with appropriate view-frustum culling, results in very cheap pixel-perfect selection.

D. Shadow Maps

Perfect shadows are computationally expensive to compute. For any given surface, we would need to calculate all of the light coming into that surface and take into account obstruction from any geometry that may any photons at any point in the scene. A path tracing solution can achieve this, but not in real time. One common technique for achieving hard shadows in a real time renderer is known as shadow mapping.

Shadow mapping renders the scene from the perspective of a light source. When rendering, we purely render to the depth buffer, discarding fragments that are further from those already stored in the buffer. This allows us to know which objects are closest to the light source. We save this buffer into a texture for usage in our later shading calculations (Figure 7).

In our lighting shader, before we do our lighting calculations, we utilize the view projection matrix of the light source to transform each fragment into the coordinate space of the shadow map. We then calculate whether the fragment we are attempting to light is closer to the light source than the fragment stored in the shadow map. If it is closer, then we know it is not occluded by any geometry, and we can light it. If it is further from the light source, we do not add any contribution to the fragment from the light source, as it is occluded by geometry. This produces an effect very similar to a shadow (Figure 8).

E. Normal Maps

Classically, surface normals are calculated by linearly interpolating between the normals of the three vertices that comprise each triangle. As this normals are used extensively in lighting calculations, increasing the normal resolution of our models can greatly increase the graphical fidelity of our rendering. Unfortunately, simply adding more vertices is not viable, as the cost to render enough additional triangles to actually make a visible difference is simply too great.

There is an alternative approach to increasing the normal resolution of our objects, however. We can use textures to
store the normals of our objects (storing $x$, $y$, $z$ as $r$, $g$, $b$) (Figure 9), using the UV coordinates of each vertex to look up per-pixel normals. This provides us much greater resolution, but poses a problem. These normals are defined assuming that we are actually in the coordinate space of the triangle we are rendering. This space is known as tangent space, as we can define the coordinate space of a triangle by its vertex-defined normal, tangent, and bitangent vectors. We can then use these three vectors to come up with a transformation matrix that we apply to all of our other vectors. From then on, we do all of our lighting calculations in tangent space instead of view space or model space, and can freely use the normals that we look up in our normal maps.

IV. RESULTS AND DISCUSSION

We rendered the same models with the same materials in both Arsee and the game engine Unity. Our engine achieve approximately approximately half the frame times Unity did (Figure 13). However, as our engine does not have quite as many graphical features as Unity, it is not valid to draw conclusions that our renderer is actually faster. It is a promising initial result, and it would be worthwhile to continue comparisons as Arsee receives further feature updates.

V. RELATED WORK

A great deal of research has been conducted in the fields of computer graphics and software engineering with regard to real-time interactive applications, most notably video games, due to their demanding performance requirements. As the nature of computer graphics programming is complicated, we separate work in the field from other work in game development.

A. Computer Graphics

The greatest challenge in computer graphics for real time applications is achieving a realistic approximation of the
rendering equation in real-time through GPU accelerated application development. Though direct lighting in a scene is reasonably cheap to compute, accounting for the propagation of light, or global illumination (GI), is considerably harder.

There are many different methods of computing global illumination in real time. Some techniques, such as radiance transfer, require precomputing information about the transport of all light in the scene and determining the visibility of all potential sources of light. Within this domain, some methods can be applied to semi-static scenes [3] whereas others require static geometry [4].

An entire subdomain of research has been devoted to instant radiosity [5], a method that approximates light propagation by bouncing energy around a scene and leaving virtual point lights at the points of collision. Individually cheap to compute, virtual point lights do not contribute much to a scene on their own, and therefore instant radiosity methods require tens to hundreds of thousands of virtual point lights to best approximate light propagation. One improvement to instant radiosity allows for the fast computation of many (imperfect) shadow maps, allowing for real time rendering [6].

Other techniques attempt to operate within screen-space, approximating the distribution of light using information only available within active frame buffers. Screen-space ambient occlusion [7] is one method almost universally deployed in modern video games, as it cheaply mimics the blockage of light by nearby geometry. Though these techniques can be quite powerful due to their low cost, their stochastic nature often leads to sampling artifacts, and require an additional post-processing step.

A relatively recent advancement introduced cascaded light propagation volumes [8] for approximating global illumination...
through the use of spherical harmonics and lattices. As this technique does not require any pre-computation and can be executed in mere milliseconds, it can be applied to real-time interactive software with dynamically changing scenes. The technique was first applied to Crysis, a video game renowned during its release for its revolutionary graphical fidelity.

B. Game Development

Video games are often enormous enterprise applications with many codependent systems operating in parallel. Developers must often neglect commonly held beliefs about good design practices to meet the strict performance requirements of a real time application. Even the most robust software engineering principles can be pushed to their limits in game development. Therefore, new engineering principles have been created to cater to the needs of the game development community.

Some researchers have attempted to link formalized game design theory [9] with game engine architecture [10]. As a games of different genres have different needs, game engines are often designed to target specific genres. Ampatzoglou and Chatzigeorgiou evaluated object-oriented (OO) programming and related design patterns in game development [11]. They concluded that OO applications can be costly to maintain and update, and the introduction of common design patterns can help to alleviate this risks at the cost of greater code verbosity and boilerplate. As games are often massive applications, this approach tends not to be sufficiently scalable.

One of the most common software design patterns in modern game development is the entity component system (ECS), as it reduces the complexity of game objects, or entities, by reducing them to a composition of components that are acted upon by independent systems [12]. Researchers found that this approach helps to reduce the scalability issues introduced by a typical inheritance-based hierarchical structure at the cost of implementation difficulty [13]. There are many methods of implementing an ECS, and given that this pattern forms the core of a game’s design, it is essential that performance be taken into account. One implementation utilizes wait-free hash maps to drastically reduce the time spent by systems reading and writing components [14].

VI. CONCLUSION AND FUTURE WORK

Arsee offers an an approach to real-time physically based rendering in Metal and Swift that tries to cater to both practical and educational use cases. There are many techniques employed within the engine that not been discussed here for brevity, but there are many more still left to be integrated into the engine.

In the future, we would like to look into making the following improvements to the engine:

- Material batching
- Dynamic shader compilation
- Reduced shader branching
- An interactive demo mode that demonstrates how techniques work in detail
- Radiosity-based global illumination
- Depth of field effect
- Dynamic Exposure-based tonemapping
- Integrated scene editor
- User interface for adding new meshes and materials to the scene

ACKNOWLEDGMENT

The author would like to thank advisors Professor James Heliotis and Professor Sean Strout for their guidance and oversight on this project. The author would also like to thank Professor Minseok Kwon for advice and feedback on the creation of this paper.

REFERENCES