CENG 491 - Design Overview Document

Group 20 - (GRT) GPU based Ray Tracing

Supervisor : Asst. Prof. Ahmet Oğuz Akyüz

Proposer : TaleWorlds Entartainment, Murat Türe

Barış Suğur, Abdullah Mert Tunçay, Zumrud Shukurlu, Batuhan Bat

I. Product Description

Our GPU based Ray Tracing API is based on the usage of both the advantages of the ray tracer rendering algorithms on shading and reflectance and the advantages of the GPU thread structures in parallel programming. So our GPU based Ray Tracer API uses latest sampling methods and acceleration structures with the NVIDIA CUDA toolkit to reach real time performance. In addition to these, this product uses masking algorithm to separate parts of the scene where ray tracing will be used and render small to medium size scenes in real time with high tier GPUs.

Our product aims to solve the famous Global Illumination problem in Computer Graphics. Global illumination is a process that simulates indirect lighting, like light bouncing and color bleeding. There are two types of illumination. Direct illumination involves the case when light rays bounce only once from the surface of an object to reach the eye. On the other hand, indirect illumination involves light rays which bounce off of the surface of an object before reaching the eye. So some surfaces are not exposed to any light source directly, but yet they are still not completely black. Because these objects receive some light as an effect of light bouncing from surface to surface. So global illumination involves simulating direct plus indirect illumination. Simulating both effects is important to produce realistic images. In addition, while we are creating realistic images, our product needs to handle this part in real time. So accelerating the rendering algorithm that can approximate to real time will be our other problem in our product.

Our GPU based Ray Tracing API involves hybrid path tracing which consists of both forward ray tracing and backward ray tracing. While forward ray tracing follows from light source to an object, backward ray tracing follows from object to the light source. The indirect lighting which is the main problem in the global illumination problem is solved by the backward ray tracing method. In backward ray tracing, an eye ray is created at the eye; it passes through the view plane and into the world. The first object the eye ray hits is the object that will be visible from that point of the viewplane. After the ray tracer allows the light ray to bounce around, it figures out the exact coloring and shading of that point in the viewplane. By using this method, we are able to handle the indirect part of the global illumination. After that we are using masking algorithm to make this calculation for restricted part of the image. While we are calculating these parts, we are using CUDA Toolkit for GPU programming. Main objective of GPU programming can be defined as making calculations in parallel by using CUDA threads. Each thread be assigned to one pixel and each of these pixels are calculated in parallel. This type of implementation gives us a major advantage as we try to make our calculations in real time.

In our product's development process, we also used Unity as the external game engine for managing several operations. These operations are scene loading, object management and basic

frame rendering. So our product is going to be a Unity-based API. But since these operations do not form the main part of the project, it gives us the chance to configure our product to any other game engine as long as it provides the functionality our product requires.

II. High Level System View

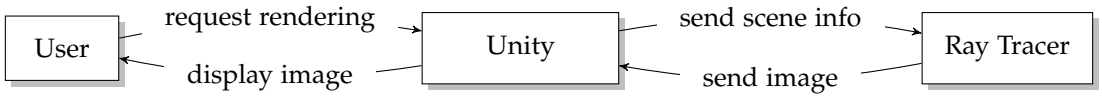

Figure 1: Context Diagram

In the context diagram above, the relationships and communications of the ray tracer with Unity and User is shown. It can be clearly seen that our implementation of ray tracer does not interact directly with the user. Instead our implementation acts as a background feature that Unity provides to user, which can be turned on and off by the user, but other than that, all relations of the ray tracer is with Unity. Since no such interface is designed for the user, all required information comes from Unity to the ray tracer as seen above. We have use some built-in functions Unity provides to get the necessary information for our ray tracer. After ray tracer finishes the execution and creates a new image, the new image is sent back to Unity to be displayed. Since our ray tracer will act like a post-processing effect, our general design is like a loop that will be executed at every rendered frame, 20-30 times per second to achieve real time performance. In order to achieve this performance, a masking algorithm is going to be used to reduce the number of pixels which rays will be generated. This algorithm will again be used with the information sent by Unity.

III. Overall Design

The end product is going to be a Unity-based API because Unity is the most used and popular game engine exist in the market. Also it can be configured to work in different game engines too as long as needed configurations are made since general principles for the game engines are similar. The three major components and their interactions can be described as follows:

• **Unity Scene**: This is the 3D virtual world that is rendered by Unity and its information is passed to the ray tracer. As any other game engine, Unity provides an interface for the scene for making users' interaction with the scene easier. Our design will work in any small-to-medium size scene that is supported by Unity.

• **Masking Algorithm**: This acts as a filter to further increase the performance of the API. The necessary g-buffers are also passed from Unity to our ray tracer. It will differ from scene to scene, hence resulting the small performance differences between different scenes. (A scene with lots of shiny surfaces vs a scene without them)

• **Ray Tracer**: This is the component where ray tracing calculations are made according to scene information and masking algorithm. This is also the last part of the design cycle, meaning that the resulting frame is computed and sent to Unity to be shown to the user.

Figure 2: Graphical Representation of Overall Design

Every information that our ray tracer needs to render the scene is given from Unity. It is not possible to interfere with Unity's own Render Pipeline, our ray tracer has to work as a post processing effect, meaning that at each frame, ray tracer will run after Unity rendered the scene. With the information about the scene and g-buffers, ray tracing will begin and end at real-time. Instead of the rendered scene from Unity, the scene modified by the ray tracer will be shown to the screen. Since our aim is to make this modification at real-time, ray tracing implementation has to be done very fast and has little impact on the performance. That is why a masking algorithm is used to filter the pixels before ray tracing begins.

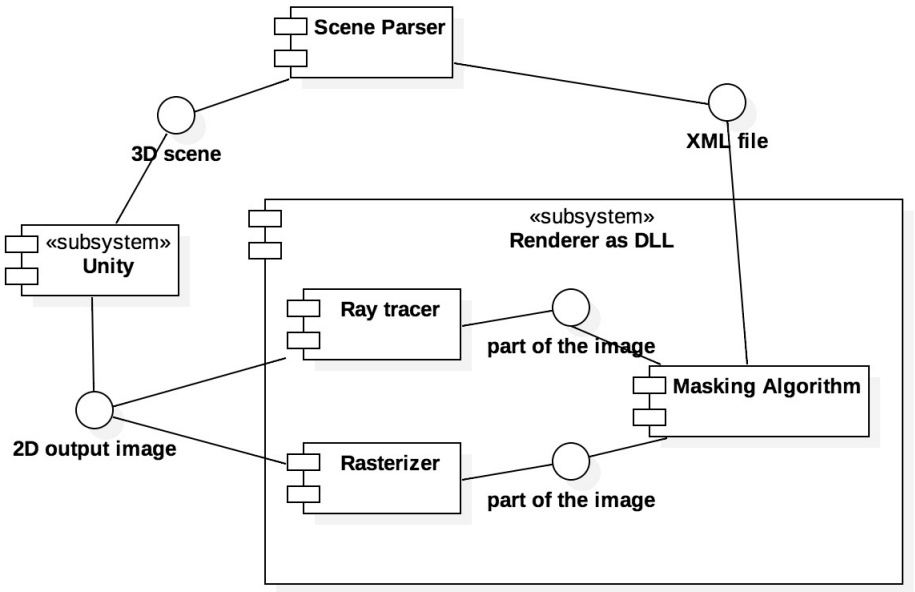

Figure 3: Component Diagram

The components of our overall design is illustrated above in the Component Diagram. Our product can be summarized an additional functionality for Unity and there are two main subsystems that interact with each other. Subsystems, components and interfaces are elaborated in the following parts.

- **Unity** : it is the principal interface that our product functions on. We obtain **3D scene** as input from Unity and display the **2D output image** in Unity again. Its native language is C#. There are options in the user interface to customize scene settings, additionally, C# scripts can be executed to make further adaptations.
- **Scene Parser** : the parser is a C# script which works as a component of Unity. It gets the necessary information about the scene through native functions of Unity for C# and writes them to an **XML file** to be sent as input to the Renderer.
- **Renderer as DLL** : It is the second subsystem of the design that renders the scene and produces the output image. It is implemented in C++ and converted to a Dynamically Linked Library (DLL) so that it can be merged with and exploited in Unity. It has the following components:
  - **Masking Algorithm** - this algorithm examines the scene and separates objects with ideally reflective materials from the rest of the scene. The first is sent into the **Ray Tracer** for rendering, and the latter is processed in the **Rasterizer**.
  - **Ray Tracer** - this component takes objects with mirror reflectance and draws them in the image.
  - **Rasterizer** - the rest of the scene is handled by rasterizer, which is a more computationally efficient tool to draw objects in the scene without fine details like ideal reflectance.

  The final output of the components of the renderer is unified and sent to Unity to be demonstrated.

IV. Alternative Design Options

In the early stages of our discussions regarding the overall design of our project we weighed the options of using OpenCL/GL or CUDA. Our primary aim has been to provide speed and ease of use from the very beginning, so we discarded the option of developing with OpenCL/GL as it would not provide us enough speed compared to CUDA since we will use Nvidia card as GPU. CUDA lets our program to use the brains of our graphics card as a sub-CPU and it's relatively simple to integrate. As CUDA is software based, much of the system must be programmed into the program's code, and thus its function can vary or be customized. Since CUDA's primary functionality lies in calculation, data generation and image manipulation, your effects processing, rendering and export times can be greatly reduced. CUDA is also great at light sources and ray-tracing. All of this means that functions similar to rendering effects, video encoding and conversion etc. will process much faster.

Another design choice was about determining the best game engine for our purposes. We were going to proceed with whether Unreal Engine or Unity 3D. After doing some research about both, we decided to go on with Unity 3D. Firstly, Unity 3D clearly comes out on top because of the number of assets in its store. It has everything, from animation and GUI generators to extensions for AI control and ORK Framework for creating RPGs. Hence, with a better asset store,

we thought that we can develop our project easier and better. Second main reason why we have chosen Unity 3D over Unreal Engine was the ease of use in terms of the editors and interfaces. Unity 3D is known for it's easy to use interface where new developers can start making games easily. Both interfaces are quite similar, with toolbars and settings within resizable and movable windows. Unreal's user interface is quite bloated and complex. It takes more time than Unity 3D in many aspects. Assets take a long time to import and save, and simple tasks require extra steps. Unity 3D is fast, and the interface is quick and responsive. That's why we have preferred Unity 3D instead of Unreal Engine.

Another design choice was about choosing the best programming language for our ray tracer and integrating our ray tracer to our project with the best approach. Firstly, we have chosen to write our ray tracer code on C++ because of the nature of the ray tracing concept being prone to Object-oriented programming itself. For this reason, among all the other languages, we have chosen C++ as the best option for Object-oriented programming related parts. Secondly, because of some speed issues, we have decided to integrate our C++ ray tracer to our project by converting whole code to a single Dynamic Link Library(DLL) file and import that single DLL file to our project as a component. We have chosen such an approach (converting C++ ray tracer to a single DLL file) over writing the ray tracer with C#, because C# scripts are compiled into bytecode while unmanaged DLLs are compiled in machine code. Hence, this would give us observable amount of speed up where we do our heavy computations while tracing all the rays.

V. Conclusion

Our product will provide a solution for a famous global illumination while providing real time performance with the help of GPU, acceleration structures and algorithms. It will give us depth of field, softy shadows and glossy reflections while approximating the performance to real time. In addition to these, it will provide a Unity based configuration for Unity users. So with the help of GPU, acceleration structures, sampling methods and external game engine, our product will give the experience of a real time ray tracer which is involved in the pipeline for the Games and Rendering applications.