A Bi-Directional Path Tracing system for rendering high quality computer images and scenes

Nathan Butt
Department of Computer Science and Creative Technology
University of the West of England
Coldharbour Lane, Bristol
Nathan2.Butt@live.uwe.ac.uk
Student ID. 16013327

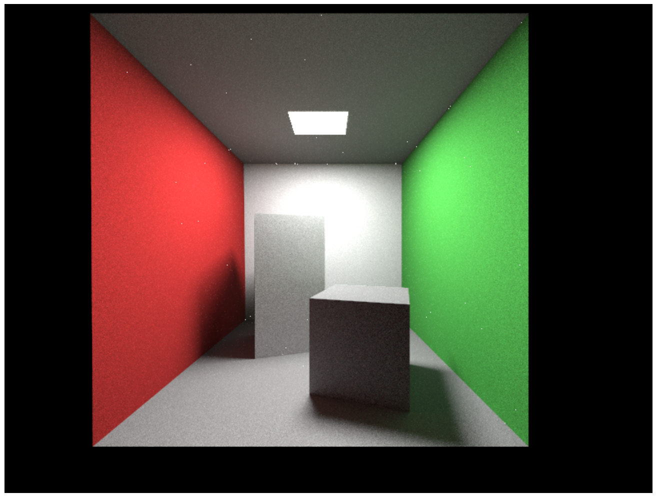


Figure 1: A sample image rendered using T-racer

Abstract

Physically Based Rendering (PBR) is a family of rendering methods built around the concept of emulating the environment as closely as possible. This approach has been a mainstay of many offline and real-time renders in recent years such as Solid Angles Arnold and Unreal Engine 4. Global illumination (GI) is a class of algorithms that is used to achieve PBR via the simulation of the propagation of indirect light throughout the scene, one such algorithm that accomplishes this is Bi-Directional path tracing (BDPT) which is used commonly in films in order to achieve photorealism. This report will detail the implementation of a Bi-Directional path tracer, evaluating the effectiveness of the algorithm in both efficiency and photorealism.

Author Keywords

Rendering; Ray-tracing; Light Transport; Monte-Carlo Rendering;

Summary

This project is a rendering system which is capable of rendering an arbitrary 3D scene using one of three ray-tracing based methods, these being path-tracing, light tracing and bi-directional path tracing. Input is taken in the form of a JSON file along with supervising assets such as 3D models and textures. Once loaded the program then continuously renders the image, progressively taking more samples of the scene until the image is in an acceptable state. These images can be saved out to either a TGA or FPM (Floating point format) file.

Biography

I am a student in games technology with a particular interest in rendering system and advanced rendering techniques. Considering the direction of travel regarding said technologies its important to gain an understanding of how it works. Therefore I intend on using this project in order to gain a wider knowledge of advanced rendering pipelines as well as gain a baseline understanding of contemporary rendering methods.

Implementation Links

GitHub Link: https://github.com/n86-64/CTP-T-racer/

Introduction

The primary objective was to implement Bi-Directional Path tracing (BDPT) with a rendering system that would allow the users to import and render custom 3D scenes with any form of 3D models. This renderer would support three fundamental materials, Lambertian diffuse, glass and perfect Fresnel mirror. In addition to BDPT the renderer should also be capable of both rendering using path tracing (PT) and a related algorithm called light-tracing (LT).

BDPT is an unbiased Monte-Carlo raytracing algorithm that is designed to model the propagation of light around a scene. Currently this algorithm is available in mainstream renderers such as Pixars Renderman [1], and is used to achieve photorealistic images. PT has recently started to gain adoption in the gaming industry with the release of real-time path-tracing pipelines such as DXR (Direct X Raytracing) [3] along with the new RTX graphics cards by NVIDIA [4]. This is a direct result of the desire of all aspects of the gaming industry to make their games more photorealistic. As such its important to gain an understanding of these algorithms as they are fairly rapidly becoming the basis of many modern game engines. Additionally, algorithms such as ambient occlusion are approximations of the methods employed here, hence understanding these GI algorithms also extends to gaining knowledge regarding contemporary methods, essential for modern game engine development.

This project is comprised of a singular application called T-racer which attempts to meet the objectives specified.

This application implements a rudimentary rendering pipeline where scenes in the form of a JSON based files are provided. These scenes configure the renderer properties such as scene name, resolution etc. along with providing details of the scenes contents. The application is able to render the specified scene using one of the three rendering methods specified with support for a range of physically based surface materials. This process involves taking continuous samples of the scene until the application terminates. These can either be rendered to a window or saved out to disk in the form of TGA or FPM file. This report will detail the various aspects of the render pipeline and critically evaluate said decisions.

Render System

The renderer was implemented utilising a linear pipeline architecture. This rendering pipeline can be broken down into four primary stages, these being Input, Assembler, Integrator and Display. In addition, there also exists two primary buffers which the pipeline interacts with throughout the process of the render. A scene buffer which contains information on the primitives stored within a scene and a frame buffer where the result of a render is written to for display.

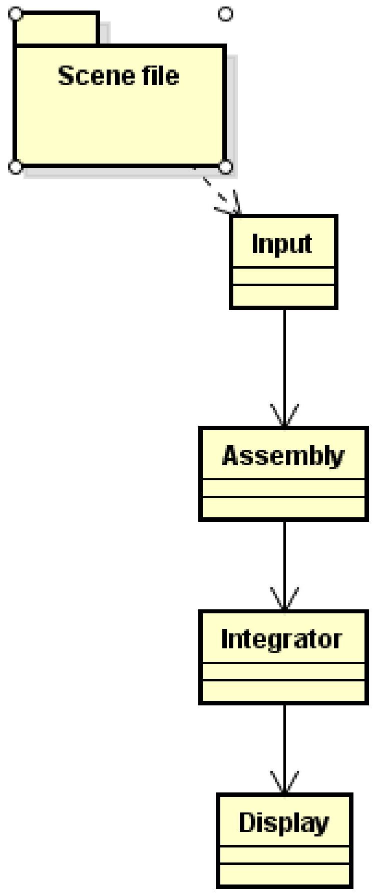


Figure 2: A sample image rendered using T-racer

When the implementation is started and a scene file is parsed, the pipeline is initiated with the input stage where all data is read into memory from the scene file. In addition, many aspects of the renderer such as the display type and display resolution are acquired from the file allowing the framebuffer and pipeline to be setup. This leads to the assembler stage where the scene is prepared for rendering via the construction of acceleration structures at which point the integrator stage is started where the image is rendered into a framebuffer using one of three rendering methods. The final stage involves writing the framebuffer to a display of some description.

This approach of a singular pipeline state is commonly used for all types of renderers due to its flexibility in configuration and simplicity of development. Examples of this approach include the OpenGL and DirectX [6] [13] renderers. This proved itself to be the case when designing the implementation as this approach allowed made the process simple with little coupling in the final implementation, reducing potential for errors.

This was implemented in a C++ application which was structured around a central object which performed the various pipeline stages. This was fulfilled by the T-racer-Renderer object with a collection of helper objects providing the various operations needed for the pipeline.

Operations

Helper Functions and core structures

Implementing these rendering techniques required a collection of additional mathematical structures such as Ray and Matrix objects for operations such as intersection testing, requiring use of a math library. This project utilised a custom math library written around the needs of the renderer specifically. This included several data structures defining Vectors, Matracies, Ray and Colour objects with a selection of appropriate operations.

An important factor when designing this library as well as considering a library was efficiency given the frequent use of these structures. An inefficient library could unnessecery lengthen render times and distort the effectiveness of these algorithms. Therefore, optimisations would be needed resulting in the adoption of SIMD or Single Instruction Multiple Data, with the classes in turn being designed in order to take advantage of this optimisation. This parallelised many of the operations in the math library resulting in significant performance improvements regarding mathematical operations, assisting in decreasing render times.

To perform Monte-Carlo integration on the rendering equation, a collection of paths are required to be evaluated. This is addressed via the T-racer-Path-Vertex class which stores light path and surface information. Making the data accessible for the integrators to perform.

Primitives

Primitives are shapes which comprise objects in the image. In addition to triangles, there are efficient intersection methods for other primitive shapes such as quads which were considered [10]. Although, it was found these primitives lack the flexibility of triangles which given in high enough number allow for spherical and non-convex objects to be rendered easily. Additionally, triangular primitives reduce the need to implement additional model primitives such as spheres as well as containing simple operations for the calculation of properties such as surface area which is essential for methods later on. Indeed the use of triangular primitives simplified the implementation in both scene construction as it unified all mesh loading and processing as well as reducing the number of mathematical operations needed for surface evaluation.

Intersection of primitives was handled using a slightly modified version of the Moller-Trumbore intersection algorithm [9]. This modification implemented the same core principles of the original algorithm but approached the calculation slightly differently with operations such as the determinant being calculated via a cross of edge vectors instead of a cross of ray direction and an edge vector.

This was a result of implementation issues regarding the default approach where clear intersections were not detected despite the algorithms implementation matching that of the original implementation. Given constraints this new method was adopted from an existing renderer and was successful in resolving these missing detection issues.

Acceleration
Acceleration is key to render efficiently via the elimination of primitives which will not see any chance of an intersection with rays being fired into the scene. This was addressed with a Bounding Volume Hierarchy which is constructed using the surface area heuristic implemented based on the research performed [11].

BVH is represented as a hierarchical tree of nodes stored within a tree object. Each node possessing a bounding box representing the portion of space and a collection of indices representing the polygons within that space. These trees are then traversed recursively until a leaf node or node without and child nodes is identified, causing the polygons to be checked, with the closest intersection being recorded for use in the wider renderer.

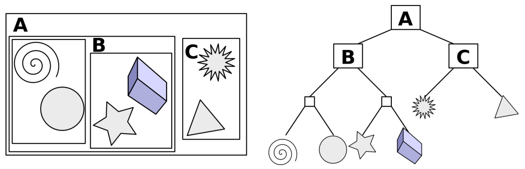
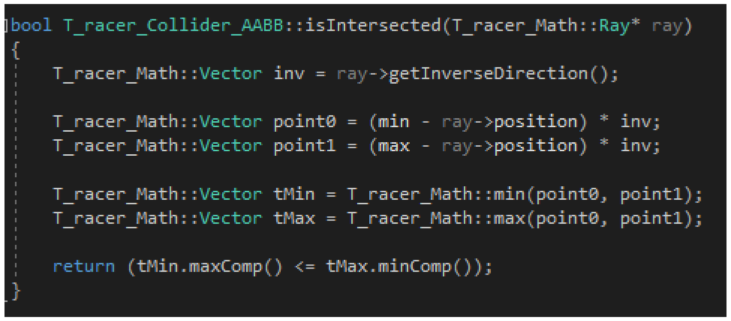


Figure 3: An illustration of the BVH concept. [12]

Upon implementation, this approach was highly effective in the renderer at decreasing the number of primitive intersection tests needed, reflecting the theoretical potential of the algorithm. These efficiencies were further improved with the slab test [8] algorithm whose implementation was based on that found in research. This not only was simple to implement but also utilised SIMD resulting in negligible impact on performance.



```
bool T_racer_Collider_AABB::isIntersected(T_racer_Math::Ray* ray)
{
    T_racer_Math::Vector inv = ray->getInverseDirection();

    T_racer_Math::Vector point0 = (min - ray->position) * inv;
    T_racer_Math::Vector point1 = (max - ray->position) * inv;

    T_racer_Math::Vector tMin = T_racer_Math::min(point0, point1);
    T_racer_Math::Vector tMax = T_racer_Math::max(point0, point1);

    return (tMin.maxComp() <= tMax.minComp());
}
```

Figure 4: An implementation of the slab test intersection [8]

However, this algorithm has a poor time complexity during the construction process with complex models comprised of a large number of polygons often taking a significant amount of time to partition. Increasing render times and preventing real-time scene adjustments. In response, an addition was made to the BVH implementation to be serialisable. This allowed pre-computed BVHs for identical scenes to be reloaded rather than recomputed, significantly reducing render times whilst not complicating the structure.

Whilst serialisation prevents duplicate work it ultimately does not improve generative efficiency. Therefore possible additions include threading for initial construction, allowing nodes to be constructed in parallel significantly reducing construction time.

Display
All colour data relating to a render is outputted to a display object which represents the framebuffer. Displays are an abstract object which represents either a window or an output file with common properties such as the framebuffer object itself. Allowing for easy adjustment of output destination whilst creating a simple interface for the renderer to interact with.

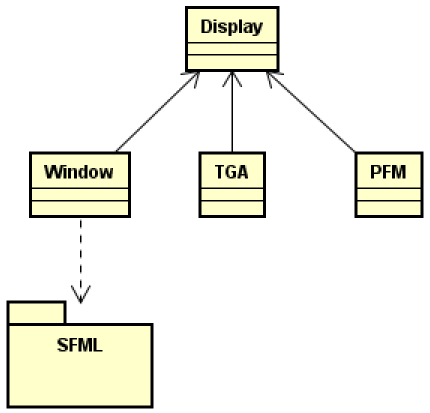


Figure 5: The system structure for displays in the renderer.

Three display objects were implemented, the first being a window display where contents are displayed to a window and two file displays which write the framebuffer to a TGA and PFM formats respectively. The window was implemented using the SFML library [2] which provided a simple interface for drawing to a window, removing the need to implement custom window and render.

One issue with the window display is the medium of display where colours may require adjustment. This was addressed with a tone mapping algorithm which adjusts the colour values for a particular display. The renderer utilises a simple algorithm where colour values were raised to an exponent.

$$f(x) = {C_{xy}}^{1/2.2} \tag{1}$$

Whilst this approach produced acceptable results, with utilisation of a GPU it could be possible to consider filmic tone mapping methods [5] which whilst more advanced produce higher quality results. Additionally, it would result in modest performance improvements with the system leveraging GPU parallelism.

Camera
The cameras is the eye into the scene, meaning rays have to be generated from each pixel of the camera in order to render an image. This is done by computing points on the image plane which can then be normalised in order to determine the direction of the ray. This creates a perspective projection into the scene.

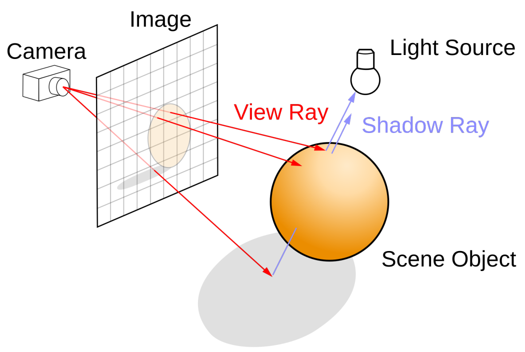


Figure 6: Illustration of camera ray generation. [7]

Initially, ray generation on the camera was performed using a series of matrix multiplications which translated screen space points to camera space and finally to world space. However, this proved to be complex and problematic resulting in a faulty camera that projected correctly but did not account for the direction of the camera.

Instead, an alternate model was adopted based on the work of Peter Shirley. This camera model did not utilise matrices and instead computed local transform vectors which is then used to derive the point on the image plane. This proved to be vastly more effective and fixed many anomalies with the camera [14]. Additionally, this method simplified implementation of both camera importance and world to screen projection which are essential methods for use in the light tracer.

Lights
Lights are sources of light energy which radiates into the scene illuminating surfaces. There are several types of lights, these include

- Point Lights: Singular points of emission in all directions.
- Spot Lights: Emission from a concentrated point to an area of the scene.
- Area Lights Emission across a specified surface area.
- Directional Lights Lighting from the whole environment from a specified direction.

All lights whilst possessing different behaviour must be capable of performing the same range of sampling and direct connection operations in order for the integrators to evaluate correctly. Therefore, lights in the renderer are expressed in the form of a single abstract class T-racer-Light-Base. This simplifies interaction with the renderer whilst ensuring the correct operations are performed.

The renderer implements both a Point light and an Area light type with the behaviour of the core functions designed around the properties of the lights. These lights implement functions to sample both points, directions and to evaluate direct light connections. Implementations of these lights were based on implementations in physically based rendering [10].

Point lights are conceptually simple and due to the singularity of light, these lights do not involve any use of random sampling as there is only one possible connection for any surface point. Additionally, as these lights emit from a singularity, these lights possess no surface normal which is an issue for the integrators.

The direction sampling function simply returns the direction towards the light position with the point sampling simply returning the position of the light. In addition to their functions, the path vertex possesses a property stating whether or not this path is derived from a point light. This allows the special case to be handled adequately without introducing any potential error. These sampling functions also because of the singular nature of the light connection return a probability density function of 1.

Area lights in principle can be comprised of any shape, however in order to reduce complexity area lights are comprised of collections of triangles which the individual triangle in turn being sampled when required. Similar to the reasoning behind the core render primitives, this allowed for flexibility in the creation of various lights. Whilst it required implementation changes, these were quite simple to make and ensured that additional shape primitives would not be needed.

Points are sampled via the use of triangle sampling which is a simple algorithm which retrieves a point using barycentric coordinates. Given light can emit in any direction perpendicular to the area light surface, its appropriate to utilise cosine hemisphere sampling [15] [10] which upon sampling a unit disks projects onto a sphere allowing the direction to be retrieved.

Materials

Materials represent the BRDF quantity that is evaluated in the rendering equation with each surface being assigned a material. As the goal is photorealistic imagery these BRDFs have been modelled on well-established physically based functions, as these adhere to some basic principles regarding conservation of energy [10].

The BRDF in an integrator needs to both define a destination direction to a surface point and evaluate the current surface point in order to account for contribution to radiance at the intersected surface point /citepbrt. This is reflected in the Evaluate and Sample Functions. These functions both evaluate the return colour of a surface and the direction of the light ray determining the next point to be evaluated. Materials also evaluate the probability density of a surface which is used to weight the surface contribution to the light path. These functions are modelled on the principles of importance sampling as stated in the research. The probability density of a surface is dependent on the BRDF model, meaning it needs to be evaluated during sampling.

Materials are implemented using the T-racer-Material class which similar to the lights is abstract due to the variety of different possible behaviours, allowing material selection to be flexible for the surface in question.

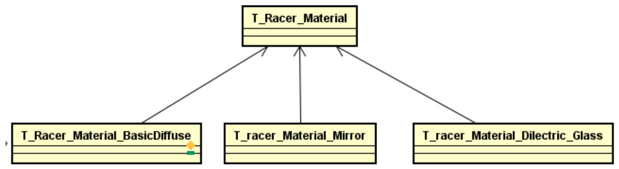
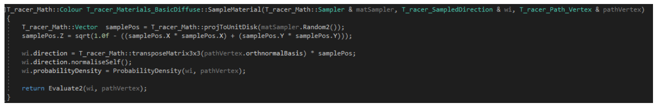

T_Racer_Material
T_Racer_Material_BasicDiffuse
T_racer_Material_Mirror
T_racer_Material_Dilectric_Glass

Figure 7: A visual representation of the structure of the material system.

Three materials were supported whose algorithms were derived from further research into physically based BRDF. The first is a Lambertian surface where light is uniformly distributed in all directions perpendicular to the surface. Evaluation and Sampling was a simple process only involving a limited collection of calculations for each.



```
T_racer_Math::Colour T_racer_Materials_BasicDiffuse::SampleMaterial(T_racer_Math::Sampler & matSampler, T_racer_SampledDirection & wi, T_racer_Path_Vertex & pathVertex)
{
    T_racer_Math::Vector  samplePos = T_racer_Math::projToUnitDisk(matSampler.Random2());
    samplePos.Z = sqrt(1.0f - ((samplePos.X * samplePos.X) + (samplePos.Y * samplePos.Y)));

    wi.direction = T_racer_Math::transposeMatrix3x3(pathVertex.orthnormalBasis) * samplePos;
    wi.direction.normaliseSelf();
    wi.probabilityDensity = ProbabilityDensity(wi, pathVertex);

    return Evaluate2(wi, pathVertex);
}
```

$$\begin{aligned} C(X_i) &= ProjToUnitDisk(\xi_1, \xi_2) \\ C(X_i).Z &= \sqrt{1 - ((C(X_i).X)^2 * (C(X_i).Y)^2)} \end{aligned} \tag{2}$$

$$\begin{aligned} pdf(x_i) &= \frac{\cos\theta}{\pi} \\ & where \\ \cos\theta &= |\vec{N} \cdot \omega_i| \end{aligned} \tag{3}$$

Figure 8: A equation and code snippet of the material evaluation

The other materials are a mirror and a glass material which are classified as Fresnel materials meaning they are capable of refraction and reflection. Fresnel materials caused issues with the implementation regarding object illumination. This was the result of not accounting for the Dirac delta distribution [10] when calculating direct lighting. The function derivative means that direct lighting should not be evaluated. This required a slight adjustment in the light path structure and the integrator object which was simple to implement and resolved the intensity issue.

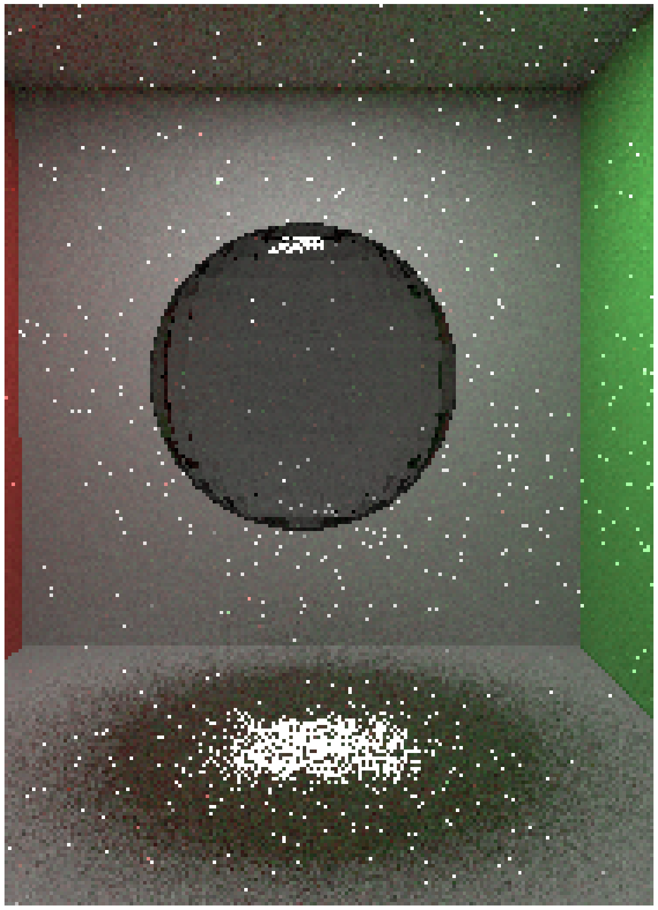


Figure 9: An example glass material

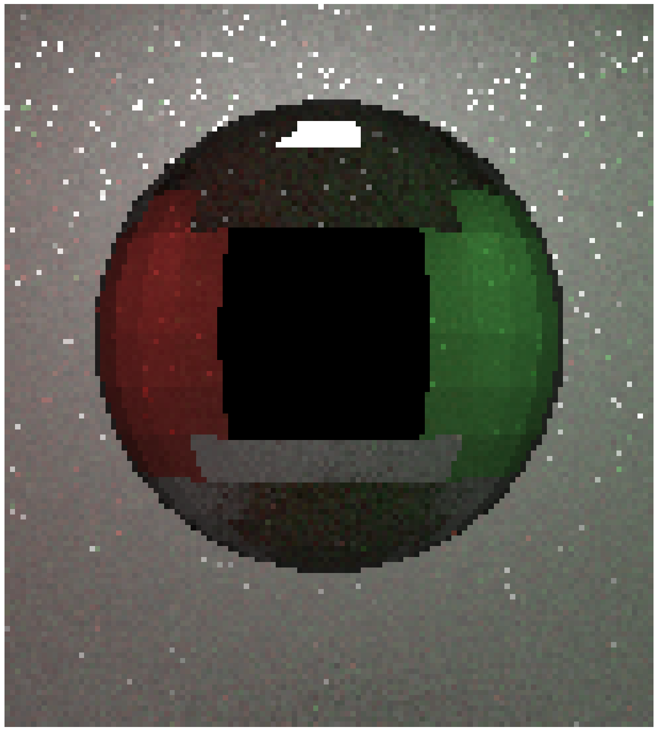


Figure 10: An example mirror material

In addition, the primitive also calculates a transformation matrix called an orthonormal basis which allows points in world space to be transformed to the primitives local space with a Z-up value. This is calculated using the Gram-Schmidt method [10]. This was an effective addition, as it allows for material sampling and evaluations to be greatly simplified due to the local frame of reference. This was greatly exemplified with Fresnel surfaces where the transform allows the BRDF to be simplified to the stated sign inversion rather than having to evaluate the Fresnel reflection equation which would have been more computationally expensive. However, this method is prone to error and can be complex to debug an implement. This was also the case with the mirror material where reflections were not transformed correctly due to a failure to transpose the matrix.

Integrators
Integrators are the core component of the renderer that performs the Monte-Carlo integration in order to evaluate the rendering equation for the given surface points. Each integrator initially constructs a light path with the BDPT integrator constructing two light paths from both the camera and the light source. A PT integrator constructs a light path from the camera with the LT constructing paths from a sampled point on the light source.

All light paths are integrated via recursive evaluation of the indirect portions of the rendering equation. This is a recursive process that terminates upon either the miss of a primitive or the failure of the Russian roulette which is used instead of fixed path termination.

Initially naive Russian roulette was implemented, however this failed to account for the luminance of the light path. Further research led to the discovery of luminance weighted Russian roulette which accounts for the contribution of the light path. This makes the termination more importance driven resulting in less noise in the final image.

Upon the construction of a light path these are then evaluated by the respective integration methods. This includes evaluation of direct lighting where light paths are connected to the light source in the case of a PT or the camera in the case of a LT. These direct lighting connections were evaluated based on prior research.

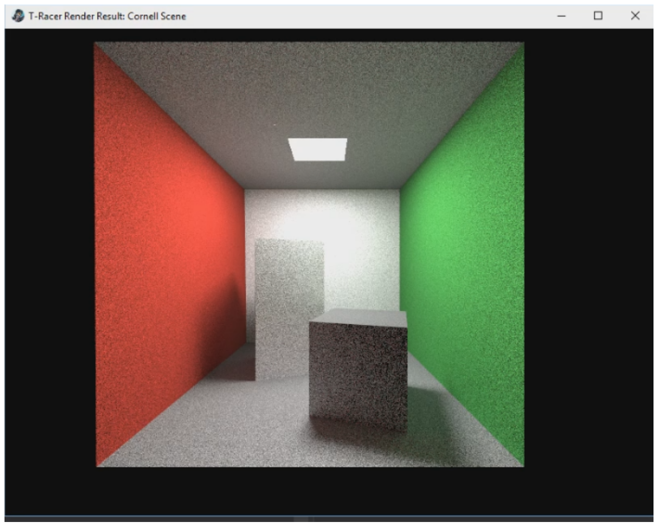


Figure 11: Render result of a light tracer

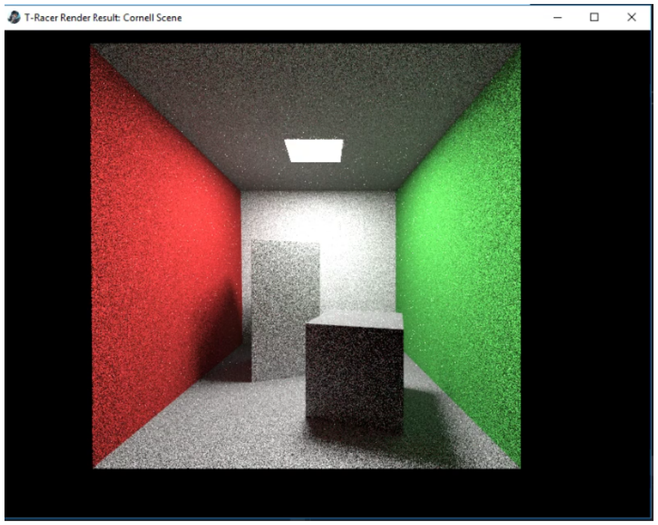


Figure 12: Render result of a path tracer

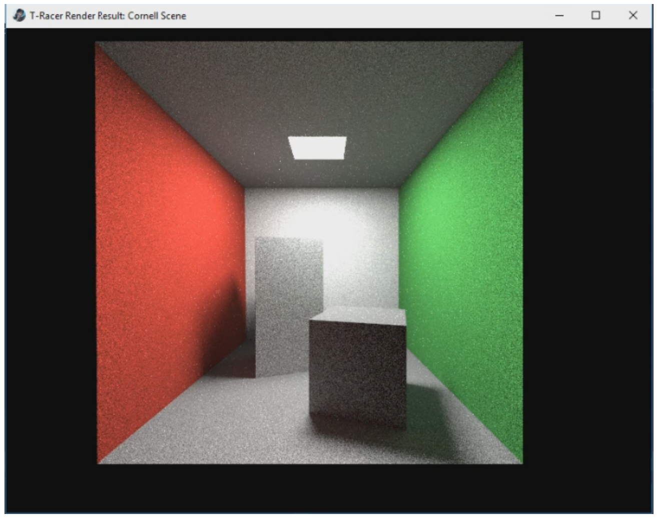


Figure 13: Render result of a bi directional path tracer.

Results and Evaluation

BDPT was implemented semi-successfully, however this implementation was limited via a number of issues regarding material behaviours and integrator behaviour. This was fairly obvious with the evaluation of glass where the caustic produced and the glass material were too dark. However, other than this issue the algorithm performed as expected.

In comparison to LT and PT integrators, BDPT is a combination of some of the strength of both methods which helps to resolve anomalies between the two methods. This is best exhibited in glass where an LT integrator is capable of rendering a caustic burn into the floor, however its unable to render the glass in the sphere due to material properties, and the PT is able to render the glass but is unable to render a caustic precisely. BDPT via the combination and connection of both paths eliminates this problem.

In addition BDPT has a faster convergence rate as shown in experiments where the image is significantly as illustrated below. Although, as theory would suggest this came at a computational cost due to the need to contribute two paths rather than one.

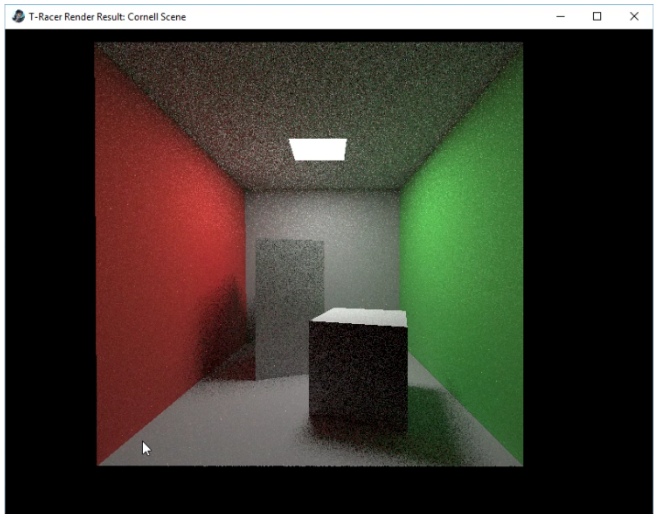


Figure 14: A 5 sample path trace render

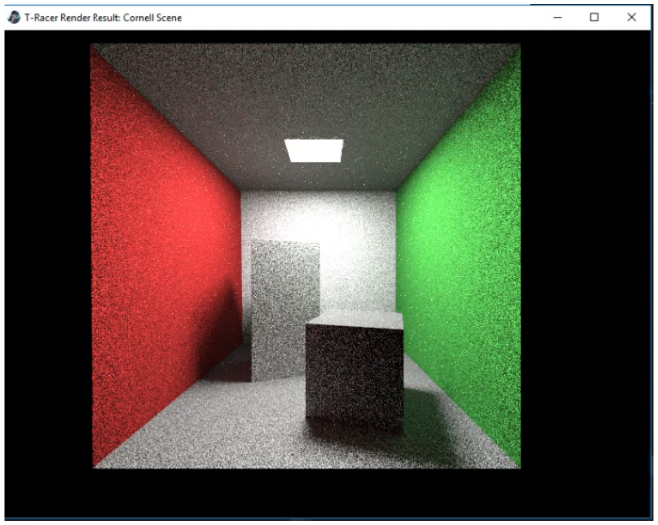


Figure 15: A 5 sample bi-directional path trace render

Overall, whilst BDPT has major advantages regarding evaluation of several materials and convergence, its computational expense can significantly increase render times. For many use cases, the use of path tracing would be more appropriate as it would be faster to evaluate. This is true especially for modern game engines looking to utilise pure GI algorithms.

Conclusions

In conclusion, the system did meet some of the core objectives and did demonstrate BDPT as well as PT and LT integrators. However the quality of the images whilst approaching photorealistic, was not to the standard that was expected at the end of the project. This is the result of issues relating to the implementation of the integrators as well as possible bugs with some of the materials. There is also noticeable pockets of noise and artefacts in the image resulting from a lack of additional importance methods. Additionally, whilst the application is somewhat configurable with scene files, it is limited in its functionality regarding user interface making it difficult for general use.

However, despite these flaws the implementation of this pipeline has served as a strong means of understanding the differences between various GI algorithms as well as the various helper processes that allow these algorithms to function. The experimentation of each of the methods provides useful information when considering application in the various industries that utilise computer graphics. Additionally, the flexible design of the system allows for easy modification meaning many of the issues can be addressed in the long term development of the system.

In the long term a number of modifications could be made in order to improve the quality of the system and its resulting renders. Firstly, the optimisation of integrators and the fixing of existing rendering bugs which would improve image quality and performance of the renderer. This could also be supervised with the addition of methods such as multiple importance sampling which would further reduce noise and artefacts in the image.

Appendix

Appendix A. Project Log

Project log can be found attached to this document.

References

[1] Pxrvcm.

[2] Sfml.

[3] Announcing microsoft directx raytracing!, Mar 2019.

[4] Nvidia rtx platform, Feb 2019.

[5] Dille, S., Fuhrmann, A., and Fischer, G. Real-time tone mapping - an evaluation of color-accurate methods for luminance compression (09 2016).

[6] GrantMeStrength. Graphics pipeline - windows applications, May 2018.

[7] Henrick. *Ray tracing diagram*. Apr 2008.

[8] Majercik, A., Crassin, C., Shirley, P., and McGuire, M. A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques (JCGT) 7*, 3 (September 2018), 66–81.

[9] Möller, T., and Trumbore, B. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, ACM (2005), 7.

[10] Pharr, M., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation (The Interactive 3d Technology Series)*. Morgan Kaufmann, 2004.

[11] Rubin, S. M., and Whitted, T. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph. 14*, 3 (July 1980), 110–116.

[12] Schreiberx. *An example of a bounding volume hierarchy using rectangles as bounding volumes.* Dec 2011.

[13] Sellers, G., Wright, R. S., and Haemel, N. *OpenGL Superbible: Comprehensive Tutorial and Reference*, 7th ed. Addison-Wesley Professional, 2015.

[14] Shirley, P. Ray tracing in one weekend.

[15] Shirley, P., and Chiu, K. A low distortion map between disk and square. *Journal of graphics tools 2*, 3 (1997), 45–52.