Demonstrating non photorealistic techniques on raytraced images

Mohammed Thanim Mahbub Rahman

***Abstract*—I created a single-threaded instance backward path tracer in C++17 with the goal of demonstrating non photorealistic effects. My adaptation of producing a raytraced scene involved manipulation of sampling rays in a pixel and how I would utilise the data collected in the pixel to produce non photorealistic effects. In this report, I demonstrate three methods of non photorealism, these methods being toon outlines, cel shading and baking brush strokes in a scene. The outline shader performed well, using a sobel operator and representing each pixel in the kernel as a ray, but had issues when dealing with reflective surfaces. The cel shader example uses the incoming ray and the angle towards the light and if the angle is greater than a threshold, the point appears darker. The technique may require further expansion. The method used for paint brushes utilises a particle system placed by sending rays into the scene and placing onto the scene based on the normal and slows down rendering due to a simple bounding volume hierarchy and can be further improved.**

***Index Terms*—Raytracing, Path tracing, Outlines, non photorealism, non photorealistic, non-photorealism, non-photorealistic, painterly**

I. Introduction

Raytracing is a form of computer graphics imagery in which the transport of light is simulated to interact with a virtual world and to interact with the environment. Due to their ability to simulate light, raytracing can produce light phenomena such as reflection, refraction, soft shadows, motion blur, depth of field, scattering, caustics, ambient occlusion, subsurface scattering and chromatic aberration very accurately [1]. Some light phenomenons such as photon and their interaction with gravity are ignored when modelling light in raytracers as they are considered irrelevant to their use case.

In my project, I aimed to successfully utilize existing techniques often found in rasterisation techniques or are not possible to perform and create non photorealistic techniques onto the context of raytracing produce a functional raytracer capable of generating non photorealistic computer-generated imagery.

I employ various techniques that are commonly found in computer graphics and apply these techniques to my raytracing application.

II. background

Ray tracing has been used in 3D animation and VFX in the film industry and graphics in video games [1]. The application of using ray tracing in the film industry has been around since 2006 [2]. Lately, raytracing has seen a rise in being used in video games (utilising realtime rendering) to produce a photorealistic look to improve the visual experience. This has been made possible by the advancements of graphics processing units (GPUs) and newer graphics application programming interfaces (APIs) which offer the ability to perform low level operations on the GPU, whilst adhering to modern hardware.

A. Non photorealism

Photorealism involves producing imagery, that aims to look as though it was taken as a picture in the lens of a camera. Raytracing is exceptionally well at performing photorealism as the nature of the renderer is to simulate the physical interaction of light against a surface. Photorealistic imagery also makes use of physically based rendering materials, which are materials which aim to accurately mimic how materials interact with light and the many phenomenons that light can perform from the interaction as mentioned before.

Non photorealism is imagery that is opposite to photorealism, where imagery is not representative of real life. Often, non photorealism is associated with stylistic renderings for artistic imagery, however, imagery that is produced from non photorealism can simply be basic in nature to communicate information to the observer. For example, imagery for technical demonstration can use high-contrast colours, ignoring shadows and outline is an example of non photorealism, but used to communicate form to the observer. In this paper, I will be focusing on using non photorealism as a way to express artistic styles often found in 2D mediums.

While the ability to cast rays and to produce light phenomena allows raytracing to generate realistic scenes, scenes do not entirely rely on the method in order to produce photorealistic scenes. They also make use of physically based rendering shaders (PBR) to provide further realism in computer imaging.

B. Popularity in films

The popularity of using non photorealism in a 3D context has had a surge of popularity in the films industry as non photorealism creates an interesting visual experience towards the audience. Knowing this, the artists producing the media are often forced to "fight" against the renderer to not produce photorealistic scenes.

Artists would also require post processing steps or texturing models to work with the intended style. This would cost time during production.

C. Justification of project

While there are existing techniques that do not require ray tracing to make artistic scenes, there are pitfalls in which ray tracers can outperform, these include:

- Artist can utilise the benefits of light phenomena such as reflections instead of preparing the scene with pre baked reflections or faking it with screen space reflections. This would open the possibility of realtime effects with reflections.
- Hardware acceleration has improved lately and can allow us to produce interesting scenes.
- Utilise artifacts in ray tracers, to provide / aid in producing non photorealism (e.g noise).
- Lack of papers regarding the topic. The topic is highly niche and not many papers were released on the topic. This may have been due to the difficulty of writing a paper or more focus being put in other aspects of raytracing as a topic.

Further benefits we can gain which are not exclusive to ray tracers alone include:

- Does not require texturing the mesh to produce the artistic style, allowing the artist to rely on the ray tracer alone to produce the artistic look.
- For outlines, extra mesh such as a convex hull would not be required, furthermore, using a convex hull often produces sharp edges or edges of the outline not being present.
- There is no requirement for a post processing step to stylise the final image.

There were also personal goals with partaking with this topic:

- Wanting to learn more about C++17, raytracers are often written in low level languages (while C++ by technicality is a high level language, it is grouped by other languages like C, Rust and Zig for their closeness to assembly code and memory management). Learning C++ would open towards more roles that often require knowledge in low level languages. I also have some experience working with C++ previously.
- The topic of computer graphics is an interesting topic and I have always wanted to learn more. As mentioned before, there has been more focus on raytracing due to improvement on accelerated hardware.

III. Literature review

A. Painterly sampling

Small paint [3] demonstrates "painterly" effect is a bug when sampling the rays. The exact bug in question is due to an incorrect usage of the Halton-series with correlating dimensions, and the errors are spread throughout the screen with an equally wrong sampling function [3].

B. Non photorealistic ray tracing with paint and toon shading

This project has goals that align similarly to mines, in which they also wanted to use the method of raytracing to produce the non photorealism of painterly effects. From their work, they were able to utilise the benefits of raytracing to perform aforementioned light phenomena. They combined toon shading (also known as cel shading) [4], which is a shading process that attempts to remove the gradient in an object and quantise between shadow or lit. This concept as a technique is an existing technique commonly found in 2D art, but can also be used on a shader in rendering 3D scenes. The project also contains an outline detection to create outlines in the scene that is rendered. The ray tracer also utilises brushes, which can be customised based on the alpha falloff. The brush is also a fraction size of the final image and is a constant size throughout the rendering process. Stroke is placed in random order to make the final image look organic. The overlap strokes are mixed via alpha channels. For future work, the project would like to utilise acceleration in their algorithm using techniques such as multithreading and more complex bounding system. They also wanted variance in the toon outline and finally coherence in the brush strokes. The images could benefit from variance of brush strokes based on transparency and size. This variance can be represented as a pressure value and as an input value to the shader. The decision of rotation in the results shown in their project is unclear if it is based on the geometry of the object or is random.

C. toon shading using distributed raytracing

In this paper [5], they only utilise the technique of cel shading and outlines as a non photorealistic technique for ray tracing. The raytracing method used is called distributed ray tracing, which performs well when creating soft phenomena such as motion blur, depth of field, translucent, e.t.c. [11]. The authors realised that there was no agreed solution to the computation of shadows with toon shading, since the often implementation is only used for local effects [5]. Their ray tracer also contains edge detection for creating outlines, the outline is a separate shader to the cel shading. Edge detection in their renderer is performed through random distribution of rays and checking against the depth buffer of the edge. If the edge was within the radius of the ray, the ray would return the area as black, else the area would be calculated with the toon shader. The paper experimented with utilising different colour spaces, and using Phong shading for calculating intensity values. The results from the project created very smooth and clean outlines. The surface of the objects in the scene are also clean and smooth due to the techniques used in the paper. The method is inefficient (with a big O notation of O(n*m) where n is number of faces and m is number of rays) when performing edge detection. The technology used was OpenGL, however the paper could find performance increase in using a modern graphics API such as Vulkan [6]. The method found issues attempting to render glass objects, resulting in blotchy results. As a reader, I do not understand the reasoning to compare the ray to a depth buffer, instead of rasterising the scene in an edge detection filter and overlaying onto the scene.

IV. Aims and objectives

For this project I aimed to provide the following:

- To provide 3 examples of non photorealism in my project which utilises raytracing (with the minimum goal of 1). This includes creating an image output for a raytracer.

- Produce a raytraced image as an output. The image can be in any image format. The image need to utilise the technique of raytracing in order to create the image.
- To produce minimal artifacts in the final render.

Further requirements that are not compulsory include:

- Import 3D models into the render scene. The model format does not need to be specific if I use an external library such as ASSIMP [7] to interpret the file.
- Optimisation of the process of rendering. This means reducing the time it takes to rendering a single frame. To test for optimisation, I will use the same parameters such as image size and ray samples.
- Utilise the renderer as an artistic process. This is to test if the rendering engine is appropriate for artists. For choosing artists to work alongside the rendering engine, I plan on picking users who have familiarity to 3D regardless of experience.
- Outputting animations (frames, which can be put together by an external video software that can turn frames into videos) furthermore, utilising phenomenon found in motion such as motion blur, potentially mixing these effects with non photorealism.
- Realtime rendering. This could create over scoping, since it would require me to switch from rendering using the CPU to rendering with a graphics card. Calculations that uses a type double would need to switch to type float, opening the issue of precision. I would also need to use a graphics API such as Vulkan [6].

V. Method

The raytracer would be a backwards path tracer. In which a ray would be casted from the camera to the scene until the ray hits a light source, unlit material or intersects with nothing, in which it will trace back and return a colour as a sample.

When the ray hits an object, it will query for the material of the object which will pass parameters into the material to get the colour, incoming ray and the scattered ray direction.

A. Initial methodology

Initially, the methodology would contain the following steps:

- Exploration on a specific technique found in the topic of non photorealism
- Understand and infer from the technique.
- Create hypothesis and experiment on techniques, which would involve combining techniques found from my literature study and trying new techniques
- Evaluate each techniques from visual results and technical results.

In comparison to the implementation, the process has some similarities. The initial plan was very naive with how much time I could spend working on each technique and underestimated the time it would take to experiment on a new technique.

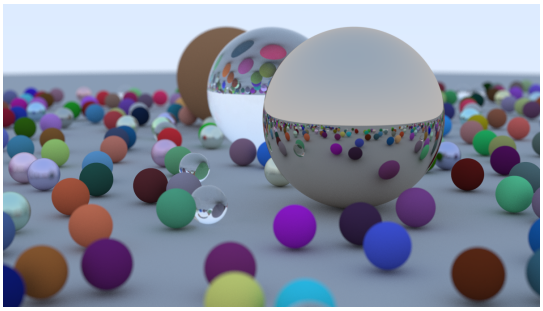

Fig. 1. Working prototype of raytracer rendering many primitives in a scene with various materials and colours

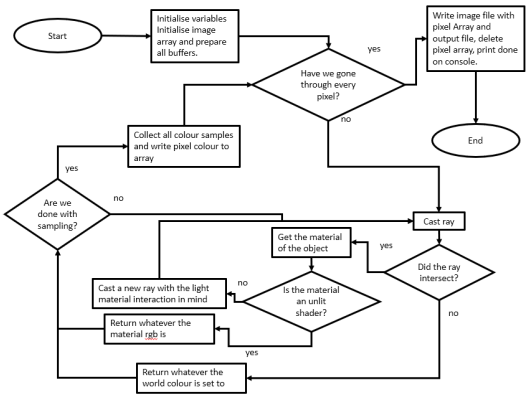

Fig. 2. System of the raytracer.

B. Prototype

For the raytracing code, I used an existing tutorial [8] [9] and used the tutorial to produce an adaption which extended the functionality of the raytracer. I would develop the project using C++17, MinGW and the GNU g++ compiler and on Win11, however the project could be compiled using any other compiler such as clang++ or MSVC. Furthermore, the raytracer code can be compiled in Linux or MacOS in their respective platform compiler. The raytracer would stay as a single threaded program since implementation of multithreading did not feel like a priority at the time. There is also the stb_image library and stb_write library, which are header only libraries for writing and reading images.

Two of the tutorials in [9] were skipped and a branch for better sampling was made following [10] but was never merged since it made implementing features already made off of the second tutorial [9] requiring a fix to support the changes made in [10].

By completing the tutorial [8] I had already achieved two of my objectives.

C. System design

As mentioned, the program is a instance backwards path tracer. The main system works similar to how a backwards path tracer would usually work. Before the raytracer starts, The scene takes in all the hittable items in the scene and

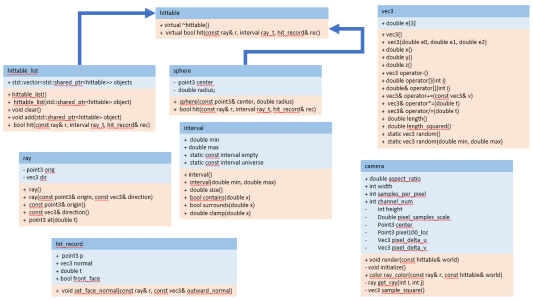

Fig. 3. Early version of the UML class diagram

generated a bounding volume hierarchy. A C-style array is initialised of type uint_8. This is what the stb library use to manipulate images. A ray is cast from a camera through to a pixel of an image. Then it will check if the ray has intersected with anything. If it did not intersect, it will return the colour of the world. Else if it hits an object, the ray will get the material information of the object that is hit. Should the material have an unlit material, the the ray returns whatever the colour of the unlit material is. If the material is a scatter based material, the material will take the ray interaction through the scatter function and the new ray is cast. This creates a recursive function which has a depth of whatever is set by the camera. This method of casting rays continuously is done as many times as the number of pixels needed. The process is repeated through every pixel. Once every pixel is rendered, an external library takes the array and outputs an image to whatever file format was specified.

Figure 3 shows an early version of the raytracer class diagram. The material function is a virtual function which has three functions, these being the emissive, scatter and rgb (rgb is for unlit shading).

Hittable objects, like sphere and quads inherits from the class hittable, which have their own implementations for information such as normals, ray intersection and such.

D. unlit materials

Before beginning experimentation on types of non photo-realistic rendering, I would need to demonstrate some form of unlit material. This involves extending the camera code to accept the unlit functionality.

Having an unlit function would also help with debugging special information about objects in the scene, such as data on normals or depth. Since the hit record (data that is acquired when a ray queries if it has intersected with a hittable item) stores the normal in which a ray intersects with a hittable object, we already have information on the normal of any surface in the scene.

Using an emission material could have also worked if I wanted to visualise debug information, however if I wanted to add debug information surrounded with other materials using materials such as Lambertian material, I would not want the ray tracer to treat the material as though it was a light source. Furthermore, unlit shaders in of itself provide as a non photorealistic material, since materials in the real world are always affected by light (absorbed, reflected, emitted e.t.c).

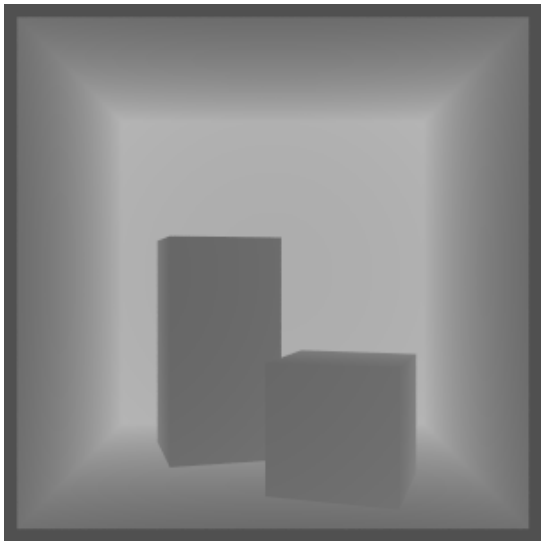

Fig. 4. Depth buffer of scene

E. Buffers

The raytracer involves extending the code to allow for buffers. These include a normal buffer and a depth buffer. Further buffers could be used to extend the ability of photo-realism. In the final project only the Normal buffer was used.

F. depth buffer

Rasterisers often come with a z buffer to let the GPU know which pixel (respective to it's mesh) goes on top of which.

Information on depth in scene is not required for a ray tracer, since rays are not concerned on ordering of meshes (As the first point the ray intersects will be interacted with removes the issue of whether the mesh is in front or behind an object) and do not require a z buffer. However, one non photorealistic technique which involves rendering outlines around the object or on the edges of an object will require information of depth on each pixel.

$$z = |P_i - P_c| \tag{1}$$

$$D = \frac{1}{n-f} * (|z^2| - n) \tag{2}$$

where f is the supposed far plane, n is the supposed near plane, P_i is the position where the point of intersection is, P_c is the position of the camera.

One issue that I found with the depth buffer was that the far plane needed to have arbitrary values. In figure 4

G. Outlines

The issue with outlines is that there needs to be context of the neighbouring area to determine if there is an edge or not. This means there cannot be a singular ray used to determine if an edge is present or not.

I would use a similar technique that is commonly found when creating outlines in a scene for rasterising. Using a sobel operator. I would then need to be able to sample normal or depth information of a scene.

The normal that is displayed should not be the actual normal of the material when looking at a reflective material. The raytracer would get a ”perceived normal” of whatever object was being perceived through the reflection. The perceived normal also assumes that the material has no roughness, which may cause the outline to not look correct in the final output if the reflective material is very rough.

An alternative method was initially proposed, where I would collect a series of distances (using the z buffer) and calculating the standard deviation of the samples to determine if there were an edge or not from the size of the standard deviation.

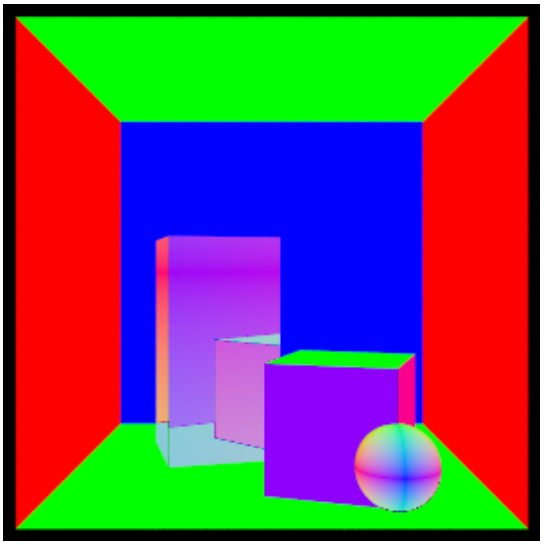

Fig. 5. The normal buffer of the perceived normal. For sake of visualisation, the normals is absolute.

Figure 5 shows what the normal buffer looks like. Note that the reflection has a slight off colouring, the cause of the issue is unknown, but it may be potentially due to how the light is collected in the end. I attempted to fix this through methods such as normalising the colour output, but the off colouring would not go. Although this could be an issue, this bug would work towards my favour since this would allow reflective materials to have their own outline instead of relying on a depth buffer to do so.

Once the material has been collected, we can run a sobel operation on the normal buffer. This would treat the normal buffer as an image and apply the filter onto the buffer. We can save the result onto another buffer as an ”outline buffer”.

Alternatively, we cast 9 rays in a grid through a pixel, arranged in a grid. Getting the normals that were perceived per point. Then we determine whether there is an edge in the pixel, by treating the 9 rays as part of the kernels. Finally, we put the output through a threshold for curved surfaces. This threshold can be manipulated to the user’s desire.

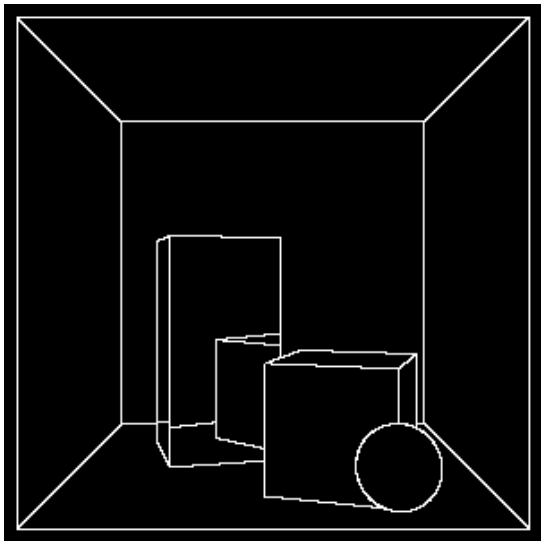

Fig. 6. 9 rays per pixel method

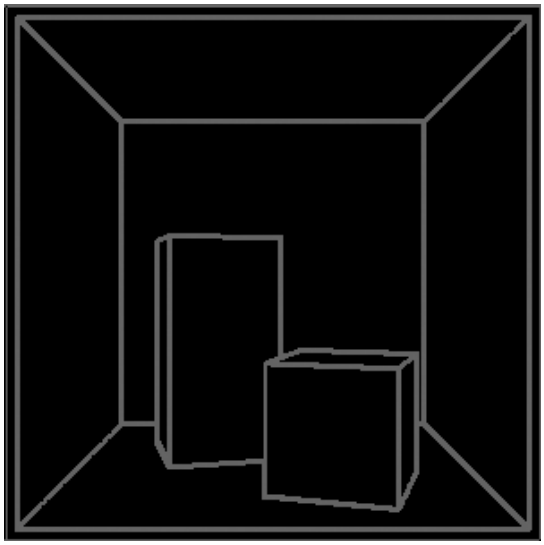

Fig. 7. Using neighbouring pixels to determine an edge

In the end, the method in figure 6 was used over the method in figure 7, since the outlines in figure 6 had finer lines. Further improvement can be made to 6 by dividing the ray grid into a 6 x 6 array. Where a pixel is separated into 4 quadrants sampling for an edge, before adding all the quadrants for pixel colour. This would remove aliasing in the outlines.

The outline buffer also has a side effect if we visualise the differnce of the normal (Figure 16)

It is worth noting that the outline shader does not work with refractive materials, since due to time constraints, an implementation was not made for the perceived normal. The outline shader also cannot do reflective curved surfaces as demonstrated in figure 17. The outline shader also does not work with motion blur.

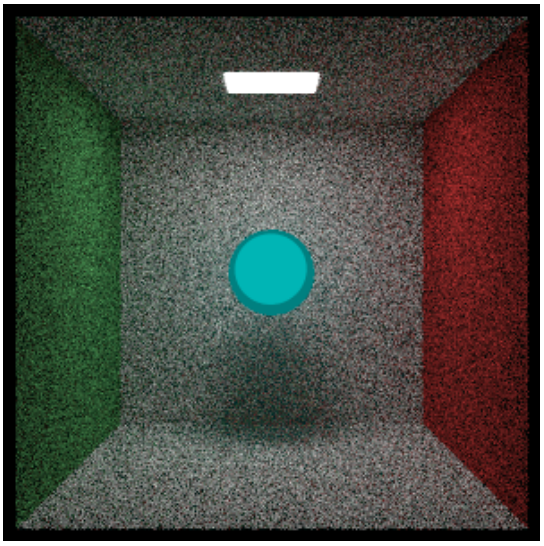

Fig. 8. Cel shading

H. Cel shading

The cel shader takes inspiration from blinn phong shading. But use the unlit material.

The cel shader takes in the incoming rays, and checks the angle the ray is entering compared to the angle of the point to the light. If the angle of the incoming ray is above a threshold defined by the user, it will retain the colour, else the colour is halved to resemble a shadow. The final result is demonstrated in Figure 8.

One downside of the cel shader implementation is the material that requires context of the position of the light in space. That means that the material also assumes that the source of light is a point light. However, this also may be useful for artists. Who may want certain materials to be affected by a specific light source. An example is character faces in animation, where the face needs to be visible even if the global lights cannot reach the face.

I. Painterly affect

In [4], they found that their method for brush strokes was extremely noisy and would change every frame. I aimed to combat this issue by ”baking” the paint strokes into the scene.

My method aims to roughly follow the techniques demonstrated in [5].

I started by first implementing a transparent material and a mix material. The transparent shader takes the incoming rays and casts them forwards as though the rays never went through anything. The mix material contains 2 materials a and b and a threshold value (if 1, it will be 100% A, if 0 it will be 100% b). when the material is hit, it will generate a random number, and depending on the value, if it is higher than the threshold it will use the second material, else the first material. This gives the illusion that the material is mixing.

I also implemented a version of the mix material which takes in an image and wherever the ray hits it will take in the UV of the point it intersects and receive the alpha value of the UV coordinate. The alpha value would be used as the threshold. This would allow me to support transparent images.

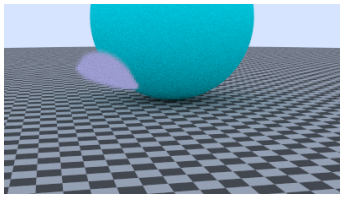

Fig. 9. Demonstration of transparent images being utilised

Fig. 10. Brush stroke used

To bake the brush strokes onto a scene, I would shoot rays into a scene in a random direction from the camera, and if the ray intersects with an object, place a plane in the point; otherwise, retry. This fundamentally acts as a particle system for the brushes. Note that baking the brush strokes onto the scene would also mean that it would not be a viable option if applied in a real-time rendering system.

Originally, the brush textures were to have various stroke sizes. They were also supposed to be generated procedurally. Due to time constraints, I was unable to experiment on this idea and had to create a simple brush stroke (Figure 10) texture to test this feature.

Initially, I tested a scene with 1000 quads as a brush stroke. However, due to the density of the scene, the scene took too long to render. The scene also returned very dark. I found an issue with rendering many quads and making sure that the quads were being placed correctly. I started with 1000 quads to see if the quads were being placed correctly and rendering the scene took approximately more than 10 hours. Figure 18 shows an example of the failed scene that contains unlit quads.

I also added a function which took HSV values and converted them into the RGB colour space. I would then choose a colour in HSV colour space and have randomly distributed brush strokes in a scene that have random colour offset by angle.

In Figure 11, the difference in values is barely noticeable. This could be due to the noise of the scene, or the world colour being slightly lit (to help with the speed of rendering). The sphere was coloured white. The brush strokes, despite being reduced, still took roughly an hour to render the scene with low sampling.

Just to ensure that the placement of the brush strokes was working as intended, I produced a debug render, where each brush stroke had a lambertian material and a random colour.

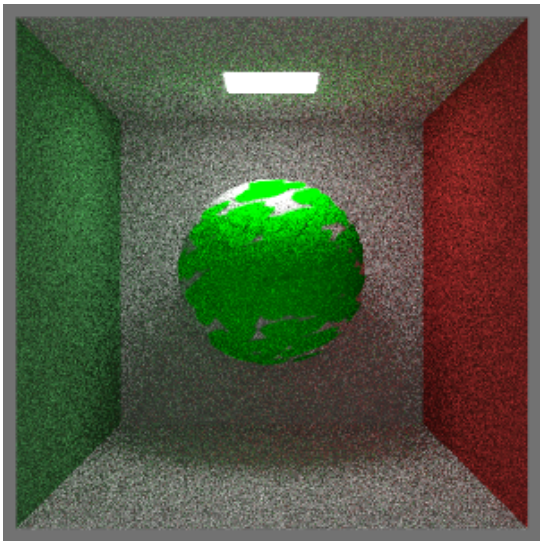

Fig. 11. In this example, there is 150 brush strokes places on a sphere. The Hue is shifted randomly by 10 degrees.

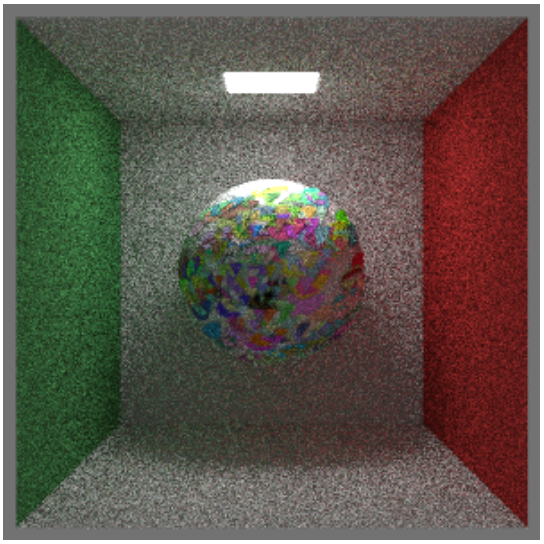

Fig. 12. A debug render. Note the quads are intersecting and it is difficult to see the brush shape

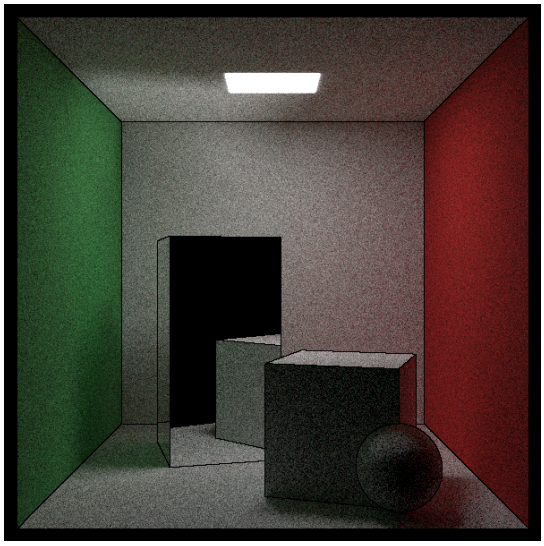

Fig. 13. Image with outlines

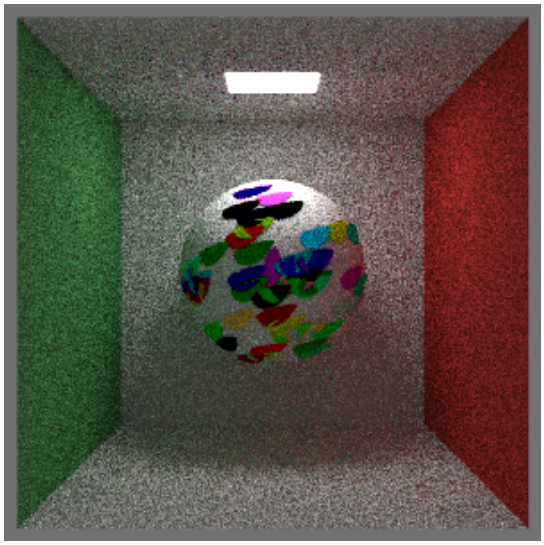

Fig. 14. Image with painterly debug, but with sparser strokes.

Figure 12 shows the result.

VI. Results

The final output (figure 13) is an example of a non photo-realistic image where an image is rendered normally, then the scene has been sampled for the edge, before being combined with the final image.

VII. Discussion

Overall, I was able to perform raytracing on my project and output an image. I was also successful in demonstrating two examples of non photorealism but was unable to provide a third example that I felt matched my expectation.

The outlines in the final image are somewhat visible but very thin, and can be quite difficult to view. The final image required a high sample to allow for the outlines to be much clearer.

The outlines also fail to encapsulate the outline of the light of the scene. This is due to the light being flush towards the ceiling of the scene (same normals, and very close to the ceiling, it can be perceived as having the same depth). This could be an issue for some artists. Querying the outlines from the samples may have worked better in my favour than querying for outlines through a singular ray, as demonstrated in [5].

The cel shading is too simplistic. It achieves the goal of cel shading, however, there are further expansions that could be made to the technique demonstrated. The lighter area of the cel shaded sphere is too big, compared to [4], where the lighter highlights are smaller. [4] also has more ”bands”, whilst my code is simplistic and halves the colour at a certain angle. I

also have not shown what the cel shader looks against a flat surface.

The painterly method system for placing on a mesh works as intended, but a system may be required to disperse the points, employing the particle system method used in [12]. The painterly strokes work better in smaller quantities with the system in place, as demonstrated in Figure 14.

VIII. Limitations

A. Hardware

Since rendering was done on a single core CPU, the time taken for rendering the images would take roughly 10 minutes. There would be times where scenes would be rendered for up to 3-6 hours to produce a scene that does not show anything.

The choice for keeping the project a single threaded program was due to limited time constraints and potential issues that may come when working with multithreading such as race conditions and deadlocks, as well as the many issues debugging that may follow. Choosing to have the project rendered on CPU was due to unfamiliarity and complexity of working with a graphical API, which could cause me to diverge from the aim of the project.

B. Time constrain

Throughout the development process, I took an iterative approach to developing the techniques presented in my raytracer. With this, raytracing is a computationally expensive task, and testing for new features would cause me to consistently wait for an output to be generated. If the technique were to fail, much time would be spent waiting for the image to generate. The wait for the renderer to finish rendering ranged from 15 minutes to 17 hours.

Working against the time constraint gave me limitations in exploring more methods for my project.

Coding my own raytracer, while offering the benefit of flexibility to implementing features, also cost time to develop. Taking an existing raytracer and trying to implement the features may have given me more time to explore more features.

IX. Further work

A. Mesh importing

Late into development, an attempt was made to add triangles as a primitive to the program. Which would open up to the possibility of adding mesh importing. I made sure to focus on implementing a flat surface and a smooth surface (quads and sphere) at the very least, but implementing triangles as a primitive for the program came very late in development and was not implemented in the end.

Adding a mesh importing feature into the renderer could show attempts of artistic test scenes utilising the materials in this report. This would also accomplish my goal of attempting to produce an artistic scene with existing mesh. As for the process of taking a mesh file into an array of triangle data. I could utilise a library like ASSIMP [7], or create my own mesh importer.

For implementing triangles, the Möller–Trumbore intersection algorithm would be required for modelling the hittables. This algorithm would allow for importing 3 points of a triangle, alongside the normals of the points.

The algorithm first calculates if the ray is parallel to the plane that the triangle rests in. A vector to represent two of the edges of a triangle is calculated followed by getting the cross product of the ray direction and one of the edge vectors. If the dot product (determinant) of the other vector against the cross vector is close to 0 (assuming we are using floating types), then we can confirm that the ray is parallel to the triangle. If it isn't, we then get the inverse determinant. We also get a vector s from the ray origin to a point in the triangle (The point that both the edge vectors stem from). Next we calculate the u value, which is the inverse determinant multiplied by the dot product of the s vector against the cross vector. if u is less than 0 and the absolute of u is greater than epsilon, or u is greater than 1 and it's absolute - 1 is greater than epsilon, then the ray has not intersect. Next we need the v value, which is calculated by first getting a cross product of s against 1 of the edge vectors. Then multiplying the inverse determinant by the dot product of the ray vector and the cross product. If v is less than 0 and absolute v is greater than epsilon or u + v is greater than 1 and the absolute of u + v - 1 is greater than epsilon, then the ray did not intersect with the triangle. The last value that is required is t, which is the distance of the point from the origin of the ray. This is calculated as inverse determinant multiplied by dot product of second edge against the cross product of s with the first edge.

B. Optimisation

There are many methods that could be done to optimise the project to render faster.

The project utilises a simple hierarchy for optimising whether a ray should ignore a primitive in a scene (Bounding volume hierarchy). There are many methods that could be used to optimise the hierarchy.

Since the project was CPU based and calculations of each pixel was performed on a single thread on the CPU. There would be a significant boost in optimisation if multithreading was adopted into the project. C++ does come with features to make multithreading on arrays more easier, if the standard library is used and C++ iterators.

Alternatively, using a GPU would significantly optimise the speed of rendering. Using existing graphical APIs such as Vulkan [6], and features such as compute shaders would allow for the project to run significantly better. However major rewrites to the code would be required to allow for this. As an example, all calculations were done on a type double. But rendering on a GPU would involve changing these types to floats, which would also introduce issues such as floating point errors. The raytracer utilises recursion when casting rays to simulate light bouncing off a material, which cannot be performed in a shader material. This can be simply fixed by converting the ray bounce code to a loop instead.

Utilising a graphical API like Vulkan would remove the benefit of making the program cross platform, since OS such as MacOS uses the Metal API. If using Vulkan as a graphical API, MoltenVK could solve this issue (an API which "translates" Vulkan commands to the Metal API equivalent when running on MacOS).

C. Denoising algorithm

The raytracer would suffer from noise unless given a high sampling rate. During development, samples were kept at a moderate size (approximately between 100 - 200 samples per pixel) to make iterations faster, the only exceptions were for rendering an image that would be the final result (example being Figure 13). Although there was a branch that was developed very late in development, which followed the tutorial of [10]. The branch allowed for better sampling of the scene and reduced the noise present in the scene. Figure 19 is one of the many renders that were done with the alternate branch. However, the materials that I implemented were broken with the newer sampling system and due to time constraints, I would not have enough time to fix and had to abandon the branch.

there are alternative methods to denoising, such as utilising shadow rays. Shadow rays are more popular in traditional ray tracers, as well as being cheaper to compute then the method I employed in the branch [10], which uses a probability density function.

One technique which can be found with NVIDIA is the use of an Ai denoising model. An Ai model could be trained to allow for a painterly affect to be produced through denoising. This would provide an issue where artists may have less control over how the final image would look like post-denoising.

creating a custom sampling technique, which tackles denoising whilst creating artistic effects could also be utilised for denoising.

D. Rewrite

Most of the code was written in the header files. This caused an issue during development with the inclusion of stb, a header-only library. Additional use of header-only libraries or other types of libraries would be much simpler if class definitions were done in c++ files instead of header files. Writing code in this style is not what I would usually do, and despite the complexity of compilation would not require me to use external tools such as CMAKE [14], it had many issues such as requiring to use inline functions in header files, making sure that functions must be const overrides and other such practices.

The practices throughout the development of the code went under the philosophy of attempting to make the idea work. Thus, the practices used were not ideal. A full rewrite would allow for better functionality while the codebase is still small. Some of the code practices demonstrated throughout the tutorial series had an entry level practice, which I ignored. This may be due to the author coming from another programming language background, or only having a shallow understanding of the language, For example, the tutorial recommended having the line "using namespace std", which is commonly a bad practice in C++ as this may introduce namespace collisions. Having all the code in a header file is another example of potentially having a shallow understanding of C++, however in this scenario I followed the methodology.

The math functions and types were followed from the tutorial, and while this was not an issue, it may have been a better choice to use an existing math library such as GLM [15], which is a popular maths library used for graphical programming. GLM also provides mathematical functions for rotations, which my current implementation of the raytracer can only do rotations on a euclidean Y axis.

The language that the program was written on is performant enough. There would not need to be a change to what language I use if I was to rewrite the program. There could be more insight into the tools that C++ offers such as the standard library array, or C++ iterators as mentioned before.

Furthermore, a rewrite could instead be expanded to making the raytracer a GPU based renderer instead of a CPU based, allow for focus on essential primitives such as triangles and demonstrate functional real-time rendering, which may make the methods demonstrated a viable option for artists.

E. Sampling

One method that did not make it due to time constraints was utilising the generalised Kuwahara filter [13] and applying it's method to how the raytracer would choose to sample colours returned from the ray. Similarly to how the outlines were performed, we could send a sample of many rays $>= 8$ and for each angle, sample the rays that are returned from that angle. we would need to convert the colour space of the rays into HSV. We get the standard deviation of each angle, using the hue (since the standard deviation of each angle would be different at every channel) and determine what the variance of the sector is through $w_i = \frac{1}{1+\sigma}$. We then multiply the variance with the average colour of the sector, before adding to the total column sum. Then we divide by the sum of the total colour.

$$K(x) = \frac{\sum_{i=0}^{7} k_i * w_i}{\sum_{i=0}^{7} k_i} \tag{3}$$

Note that the distances of the points relative to the centre must adhere to Gaussian weights.

F. User interface

The scenes were all hardcoded before rendering. The workflow would involve slightly modifying something in code, then compiling before running the renderer. The last two part of the step were ran through a batch file, which provided minimal speed to development.

Most rendering engines when working with materials will give users a node graph, which allows them to work with materials easily. There is also a viewport to communicate to the user where every object in a scene is, such as hittables, the camera position and direction, the scale of the hittables and the world position.

I can use an existing graphical interface library or create my own interface for the raytracer. Alternatively, I could use a game engine and code a viewport and interface for my rendering engine, and when I want to render an image, a configuration file is created and fed into the executable.

G. Further work on outlines

As mentioned in the discussion, the lights in Figure 13 do not receive an outline because the quad with the emissive material having the same normal as the ceiling and the depth buffer cannot perceive the ceiling because the height of the light is flush against the ceiling.

To combat this issue, another buffer can be used, which gives each hittable item a unique value. Hittables that are added to the scene can have unique value. This would ensure that hittables that are grouped by multiple hittables are not mistaken for being a different item.

In Figure 13, the outlines in my image, in comparison to the raytraced images shown in [4] and [5], where the outlines are thicker. A parameter to control the thickness of the outlines would help the outlines be more visible in the scene.

Outlines would also need more work when creating outlines for rough but reflective surfaces. As well as an implementation for outlines to work on diaelectric materials.

H. Further work on cel shading

Since the current implementation of the cel material is basic, there are many improvements I can add to the cel shading.

In my implementation of the cel shader, there is no parameter to change the threshold. This would help in making the material react more or less towards the light.

The colour quantisation method is extremely basic, by only darkening the colour of the material if it's beyond a threshold. Better color quantisation techniques could be utilised.

The cel shading material was never tested against a flat surface and would need to be evaluated against when working on the cel shaded material again.

I could also rethink how the cel shading material works by instead looking into the scatter based function instead of using a unlit shader. Rethinking the system to instead only affect the rays that is reflected off of the material to make the material look simplified. I could also instead add another material function that requires information of an array of light positions and what type of light they are. However, by implementing this technique, it is possible this strays away from the justification of the project as a rasteriser would work better.

I. Further work on painterly brush strokes

There are various improvements that can be made to the baking of brush strokes on a scene. Currently, the brush strokes provide no rotation to the strokes (rotation along the Y-axis is supported, but not for the other axis). We could have brush strokes rotated to an angle as desired by the artist.

All brush strokes are uniform. Since we have access to an outline buffer, we could firstly provide functions such as min and max random stroke size, min and max random stroke lengths for the brush strokes and size of hue difference.

The procedural generation of paint brushes utilising mathematical functions and noise could be utilise to produce a more artistic look of the painting. Adding an alpha fall off to represent the brush stroke pressure could also be utilise. Procedurally generating a unique brush texture and applying the texture onto the quad may require an external library to make this feature easier.

Importing existing brushes could also be utilised over generating brushed to allow the artist more control on the look of the strokes.

One limitation of the current feature is that the brush strokes can only move in a straight direction. If artists wanted the flow of the strokes to have variation (such as a wave like path). Motion like this would allow for variance in the scene or to help accurately model looks such as fur and hair.

One other implementation could involve having the context of the scene. Firing a ray at a random direction, and should the ray hit something, create a struct to represent the point as a progression of a brush stroke. The struct could include information such as the point in scene, the normal of the point, the weight, and the size. We can push the point into an array or pushed into a brush object. Then move the ray against the desired angle that the artist would want, firing a ray in that direction. This step would be repeated many times until the stroke length is at it's maximum or the ray hits nothing. The weight of each point and size would then be readjusted and rejected (such as brush strokes that are too short) from running a method like "stabilise". Curves to describe the pressure and weight to aid the artist. There would be issues such as too many quads in the scene slowing down the time for rendering, which would force better methods of checking for ray intersections between hittables (such as a better bounding volume hierarchy system).

The particle system method for placing brush strokes into a scene utilises rays hitting objects onto a scene and placing the brush stroke onto the point. However, the particle system cannot place brush strokes in areas that are obstructed, which may be seen if a reflective material was behind an object. A better particle system that emits from the centre of the object would solve this issue, which was the method that was demonstrated in [12].

X. Conclusion

The project has given me greater insight on raytracing as a topic of computer graphics. I have a better understanding of what processes happen when a renderer renders a scene.

I would have also picked my objectives differently, which would include the following:

- Be able to perform raytracing and to output an image in any form.
- Create a sphere object, quad object and a triangle object
- demonstrate a non photorealistic technique that can be performed in a realtime rendering system.
- demonstrate a non photorealistic technique that can be performed in an instance based rendering system.
- demonstrate a third method of photorealism.

The addition of triangular primitives into the list is an oversight on how crucial triangles and mesh can be to demonstrating the usefulness of a technique. In other reports [4] [5], the authors of their renderer had mesh importing into their program, which helped visualise the effectiveness of their method on complex geometries.

It may have been a better choice to focus on producing one technique over producing many various techniques, which may have been a factor that worked against my time budget when developing the project.

The outlines method presented has real applications on both realtime rendering system and instanced based systems, as well as the cel shading method. But the painterly method may not perform well on a realtime rendering system, unless there is optimisation to the method.

From the many challenges I faced when tackling the project, I was satisfied with the output of the outlines that my raytracer could produce, but not with the cel shader nor the painterly rendering. Despite this, I was able to demonstrate non photorealism in my application, as well as conceptualise a few techniques that could be used despite not having the opportunity to attempt the methods.

One goal I had alongside the creation of the raytracer was to demonstrate these techniques could be applied to creative applications where raytracing is already utilised. While my raytracer could potentially be used in films and animation, there would be many changes required for the raytracer to work in real-time raytracing applications such as video games. This would involve heavy optimisation of the existing techniques.

One main issue I found performing any form of non photorealism in raytracing is a singular ray is not enough to perform the effects. Instead to produce non photorealistic imagery, there should be focus on how rays sample the colour data into a pixel. There comes a challenge with the concept of raytracing. Since raytracing is supposed to be modelled after light in the physical world, trying to perform non photorealism requires rethinking on if light were to interact with these materials, how can this interaction be modified to behave incorrectly, and artistically.

One of the other challenges of this project was the lack of results that could be quantised. I was operating under whether the output of the images looked visually appealing or not.

As of the personal goals that I set for myself, I was able to gain a better understanding of the C++ language, understanding how the C++ toolset works. My better insight on the language has allowed me to critique code practices and able to identify why such practices are not preferred. I would also look forward working in C++ for future personal projects.

Finally, there are many other non photorealistic effects that could be performed does not need to be in the subset of creating artistic imagery. For example, rays are currently represented as straight lines, but creating raytraced scenes that use non linear paths (Simulating non euclidean worlds) and could generate interesting imagery or experiences.

References

[1] Ray tracing, Nvidia developer - https://developer.nvidia.com/discover/ray-tracing (accessed Nov 2024)

[2] Ray tracing news "Light makes right" July 20, 2010, Volume 23, No. 1 - https://www.realtimerendering.com/resources/RTNews/html/rtnv23n1.html (accessed Nov 2024)

[3] Smallpaint: A Global Illumination Renderer K'aroly Zsolnai-Feh'er https://users.cg.tuwien.ac.at/zsolnai/gfx/smallpaint/ (accessed Nov 2024)

[4] Non-photorealistic ray tracing with paint and toon shading https://dl.acm.org/doi/10.1145/3450618.3469173 (accessed Nov 2024)

[5] Toon shading using distributed ray tracing by Amy Burnette and Toshi Piazza - https://www.cs.rpi.edu/ cutler/classes/advancedgraphics/S17/final//projects/amy//toshi.pdf (accessed Nov 2024)

[6] Vulkan, Khronos group https://www.vulkan.org (accessed May 2025)

[7] Open Asset Import Library https://github.com/assimp/assimp (accessed May 2025)

[8] "Ray Tracing in One Weekend." https://raytracing.github.io/books/RayTracingInOneWeekend.html (accessed May 2025)

[9] "Ray Tracing: The Next Week." https://raytracing.github.io/books/RayTracingTheNextWeek.html (accessed May 2025)

[10] "Ray Tracing: The Next Week." https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html (accessed May 2025)

[11] Distributed ray tracing - https://dl.acm.org/doi/10.1145/964965.808590 (accessed Nov 2024)

[12] Barbara J. Meier. 1996. Painterly rendering for animation. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH '96). Association for Computing Machinery, New York, NY, USA, 477–484. https://doi.org/10.1145/237170.237288 (accessed May 2025)

[13] G. Papari, N. Petkov and P. Campisi, "Artistic Edge and Corner Enhancing Smoothing," in IEEE Transactions on Image Processing, vol. 16, no. 10, pp. 2449-2462, Oct. 2007

[14] CMake https://cmake.org (accessed May 2025)

[15] GLM https://github.com/g-truc/glm (accessed May 2025)

XI. Appendix

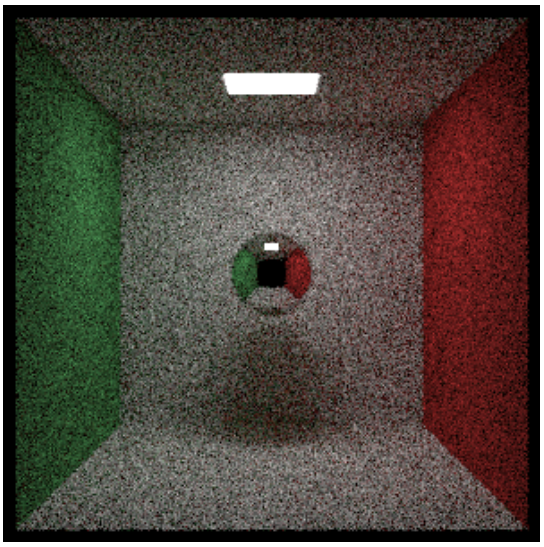

Fig. 15. Raytraced scene of the reflective ball

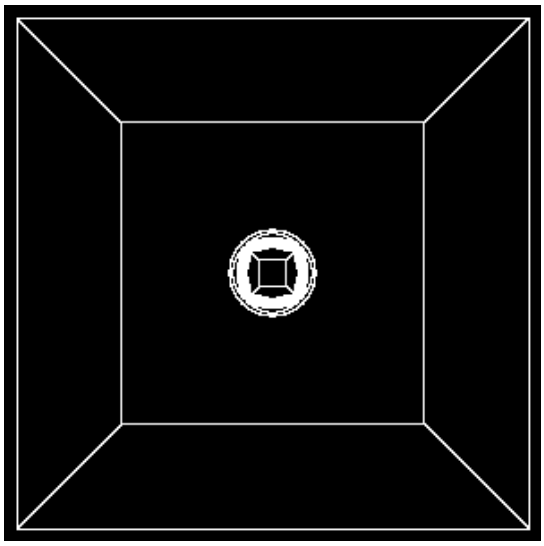

Fig. 17. outlines on a reflective ball, original scene is Figure 15. There are artifacts as the surface is more perpendicular to the rays.

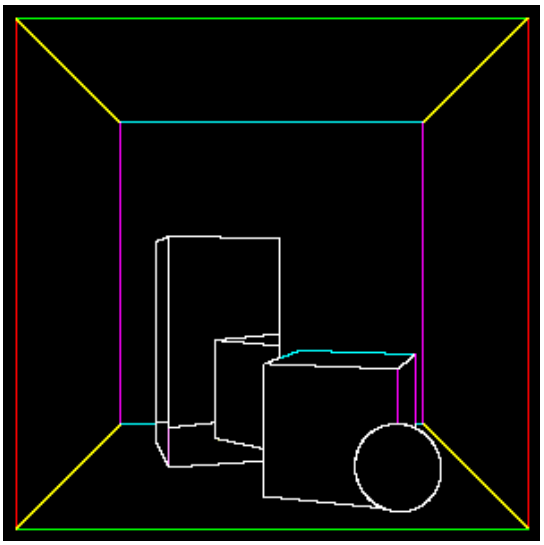

Fig. 16. If we write the difference of the normals from the sobel operation, we can see the direction of the outline.

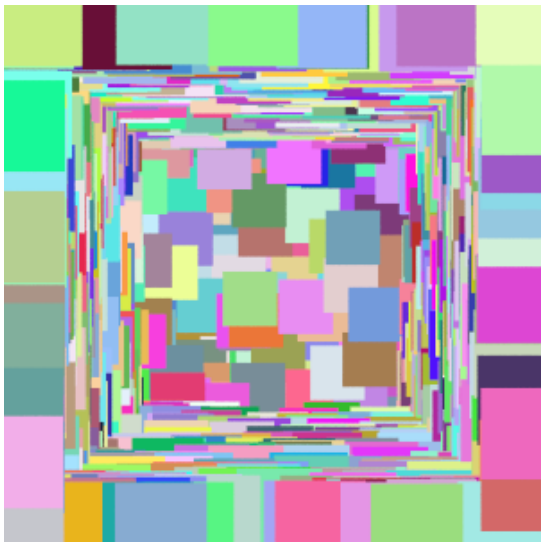

Fig. 18. Failed scene of attempting to place quads in a scene. There was 1000 quads in this scene and it took roughly 10+ hours.

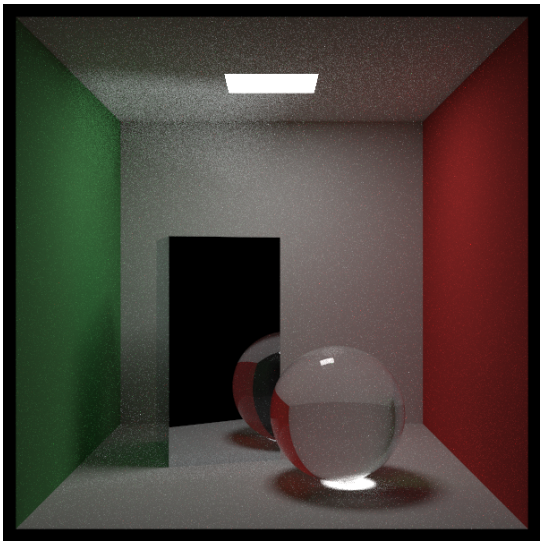

Fig. 19. One of the renders from the better sampling branch