Photon Mapping

A C++ path tracing renderer that implements photon mapping for global illumination

Matt Buckley
University of Bath

Introduction

Objectives

The objective given for the project was to render a high quality image of the Stanford Bunny using Photon Mapping. In order to do this effectively, a number of objectives were set:

The system must be able to

- Efficiently render 512x512 images
- Super sample when rendering to remove aliasing artifacts
- Utilise MERL BRDFs
- Generate Photon Maps
- Generate Caustic Maps
- Interpolate triangle normals for smooth objects
- Render refractive and reflective materials
- Render soft shadows using area lights

Why is Photon Mapping considered an advanced topic?

Photon mapping is an advanced topic, due to the complexity of the process. It is split into two distinct phases, each represented by complex mathematics in the original paper (Jensen 1996). The implementation required complex data structures to be implemented effectively (Jensen 2004), which are difficult to implement themselves. To get a nice looking resultant image, the variables of the algorithm must be fine tuned, and the iteration frequency is low due to the long render times. A certain amount of artistic direction is required to achieve high quality results.

Techniques

Photon Mapping

Building the Photon Map

Photon mapping is a two step Global Illumination technique that simulates the way light physically travels around a scene. In the first step, a photon map is generated, which contains a set of recorded Photons, distributed throughout the scene. Each recorded Photon has a Position, Incident Direction, and Flux. To calculate the map, a large number of photon rays are generated at a light source with a uniformly random direction:

```
float x, y, z;

do {
      x = (*distribution)(*generator);
      y = (*distribution)(*generator);
      z = (*distribution)(*generator);
} while (pow(x, 2) + pow(y, 2) + pow(z, 2) > 1);

direction = Vec3(x, y, z).normalized();
```

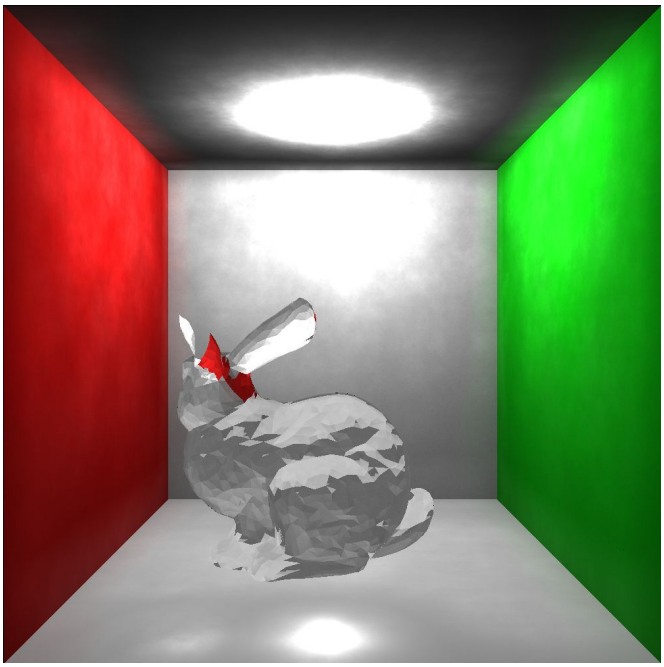
The light's power is divided equally between all photon rays as flux. The photon rays are then traced around the scene, propagating flux when they interact with surfaces. When a photon collides with a surface, there are three possible outcomes, and the technique treats each differently (Jensen 2004; Jensen 1996).

Diffuse Reflection - The photon collision is recorded in the map and the ray is reflected. The flux of the reflected ray is multiplied with the BRDF of the surface.
Specular Reflection - The photon is specularly reflected or refracted by the object depending on it's material properties. The flux is unchanged and the collision is not recorded.
Absorption - The photon collision is recorded in the map and the ray is not reflected. This is the end of the ray's path.

In order to determine the outcome of a collision, a technique called russian roulette is used. The material of the intersected surface contains probabilities for diffuse reflections D, specular reflections S, and refractions R. A uniformly random number P between 0 and 1 is generated and used to calculate the outcome. If $P < D$, it is a diffuse reflection. If $D < P < S$, it is a specular reflection or refraction, depending on the refraction probability R and a new random variable. If $P > D + S$, then an absorption will occur. This technique allows all photon rays to have a large flux, preventing any wasted work. If 500 photon rays hit a surface witch a diffuse reflection coefficient of 0.5, 500 photons with half flux or 250 photons

with full flux could be diffusely reflection. The latter is much more efficient and produces better results, as an even distribution of flux in the photon map results in a smoother image. (Jensen 2004)

Direction masking is often used to ensure that photons are not wasted going off into the wrong directions (Jensen 2004). I opted not to implement this, as my light is inside a box so photons are unlikely to be wasted, and I place my light close to the top of the box to create a nice light bleeding effect, so it looks as if light is bleeding in from the ceiling:



Light bleeding in from the ceiling is actually reflected photons from the light source just below it

Photon maps are exceptionally effective at generating Caustics(Jensen 1996), the projection of light waves onto a surface, that have been refracted or reflected by a specular object. To do this, a separate photon map is generated, specifically holding caustics. Only photons which have been through one or more specular transmissions are stored in this map. The initial photon rays are directed towards caustic causing objects, to avoid wasted work.

Low-density photon map of a white box with a gold metallic right wall. The gold surface scatters gold all over the white walls.

The recorded photons are then stored in a KD-Tree to speed up the rendering process. KD-Trees are binary search trees which are especially useful for nearest neighbour searches which will be relied on heavily in the rendering process. KD-Trees organise data based on their physical positions by splitting each level of the tree by the largest dimension of the bounding box of the data. The median point in the set is used as the split point at each level of the tree. This means that the nearest neighbour of a search point can be found in O(log(n)) time. (Foley & Sugerman 2005)

Rendering with the Photon Map

Global Indirect Illumination

In the rendering phase, the photon maps can be used to calculate the indirect illumination at a surface point. When a ray from the camera intersects with a surface, the indirect illumination at that point can be calculated by retrieving the N nearest photons to that point from the map. The radiance of each photon is then calculated by multiplying surface's BRDF (using the photon incident direction and the viewing ray) with the photon's flux. These radiance values are then summed for each neighbour, and divided by the area of the circle containing the photons, which are assumed to be lying on a flat surface.(Foley & Sugerman 2005; Jensen 2004) This gives the radiance estimate from the photon map at the intersection point. The result can then be added to the direct illumination component (the surface's BRDF using the light direction and the viewing ray) to calculate the total radiance estimate at the intersected point.

```
Vec3 Scene::getPMComponent(IntersectionContext context, int numberOfPhotons, PhotonMap* map,
bool filter, float filterConst)
{
	auto photons = map->findNearestNeighbours(*context.intersectionPoint,
numberOfPhotons); //Priority queue

	const float radiusSquared = (photons.top()->position -
*context.intersectionPoint).squaredNorm();

	Vec3 pmapContrib = Vec3(0, 0, 0);

	while (photons.size()) {
		const Vec3 inVec = photons.top()->incomingAngle;
		const Vec3 photonBRDF = our_getBDRF(-inVec, *context.outboundVector,
*context.surfaceNormal, context.material->brdf);
		Vec3 thisContrib = photons.top()->flux.cwiseProduct(photonBRDF);
		const float thisDist = (photons.top()->position -
*context.intersectionPoint).squaredNorm();
		if (filter) {
			//Apply cone filtering
			thisContrib *= (1 - (thisDist / (radiusSquared*filterConst)));
		}
		pmapContrib += thisContrib;
		photons.pop();
	}

	const float multiplier = 1 / (M_PI * radiusSquared);

	return pmapContrib * multiplier;
}
```

Caustics

Caustics are calculated using a separate caustic photon map, containing only photons which have been specularly reflected. The same process as above is used and then this result is filtered using a cone filter, which weights neighbours based on their distance to the search point, to sharpen the resultant caustic. (Jensen 2004)

Caustic caused by sphere

Optimisation

General Practices

Throughout the code, an effort was made to consistently make use of efficient library functions, rather than writing new implementations of methods. For example, the nth_element c++ standard library function was used for sorting arrays of pointers when building KD-Trees. This function sorts arrays only until the nth element is in the correct place and all elements to the left are smaller and all elements to the right are larger. Using this over a sort function for the arrays when building KD-trees is over twice as fast. (Jensen 2004; Anon n.d.)

Efforts were made throughout the code to cache and re-use calculations as much as possible, as at Triangle midpoints and bounding boxes:

```
Vec3 Triangle::getMidpoint()
{
	if (!this->midPointCalculated) {
		Vec3 midPoint(0,0,0);

		for (int i = 0; i < 3; i++) {
			midPoint += this->getVertex(i);
		}

		midPoint /= 3;

		this->midPointCalculated = true;
		this->midPoint = midPoint;
		return midPoint;
	} else {
		return this->midPoint;
	}
}
```

Threading

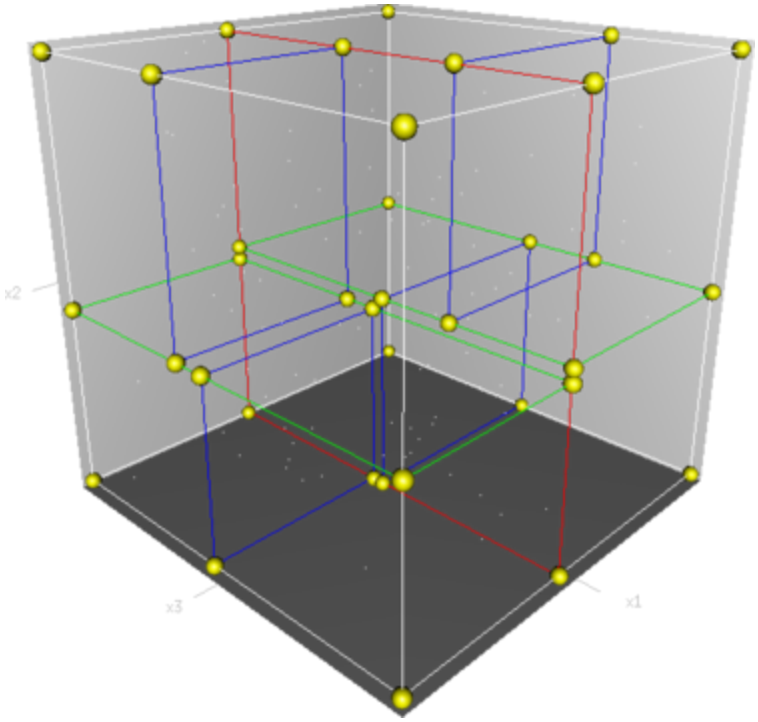
Raycasting is an inherently parallelisable problem (Reinhard & Jansen 1997) due to the fact that the scene data remains unchanged throughout computation. All results are written to the render target. This means that it may be easily parallelised, without the overhead of locking access to certain variables. I chose to thread my main render function, and my photon mapping function, which were both very time consuming operations. The key issue to account for is that rendering work is not evenly distributed in image space. Half of the image may contain millions of triangles and the other half may be empty. The rendering work must be divided equally between threads, so I chose to interlace the threads. Each thread knows it's thread ID and the total number of threads, and it iterates over all columns in the image, starting at it's thread ID, and incrementing by the number of threads each time. This resulted in a huge speedup, and 100% CPU utilisation when rendering.

Given more time, I would have liked to attempt the parallelisation of KD-Tree balancing, as this was another time-consuming operation. I'd also like to implement my ray tracing algorithm in Nvidia CUDA, to make use of the hundreds of CUDA cores in my laptop's GPU. (Popov et al. 2007)

A partially finished render, showing the thread interlacing

Mesh KD-Tree Hierarchical Bounding Volumes

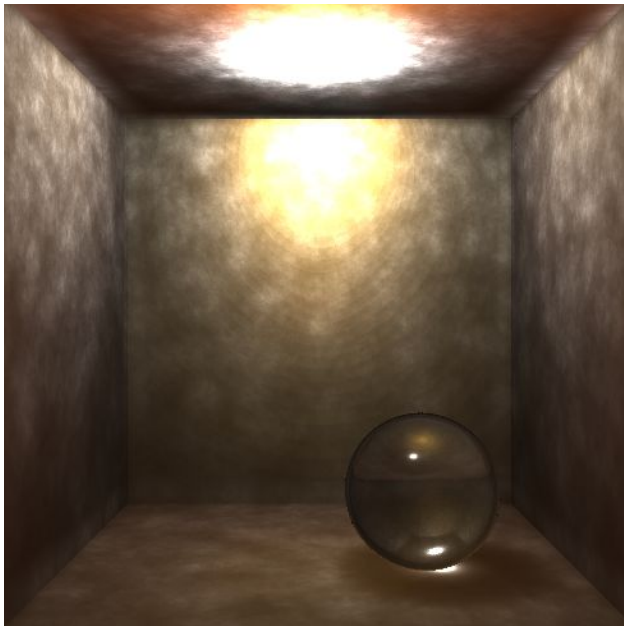
The KD-Tree data structure previously mentioned for storing Photons was adapted to hold Meshes. Each level of the tree holds a list of Triangles and a bounding box containing the triangles. When intersecting a ray with a mesh, the ray first intersects with the top level bounding box. If there is a hit, it tried to intersect both children. It does this recursively until it reaches a leaf node where it will test all of the triangles at the leaf. The 80k triangle bunny can be rendered in just a few seconds using this method, as the tree is only 8 nodes deep. (Foley & Sugerman 2005)



KD Tree visual representation from wikipedia

Bidirectional Reflectance Distribution Functions (BRDFs)

BRDFs are functions that output the colour of a material, given the incoming and outgoing ray angles. They can be measured physically, and there are large databases of physically measured BRDFs available online.



Gold metallic paint BRDF from MERL

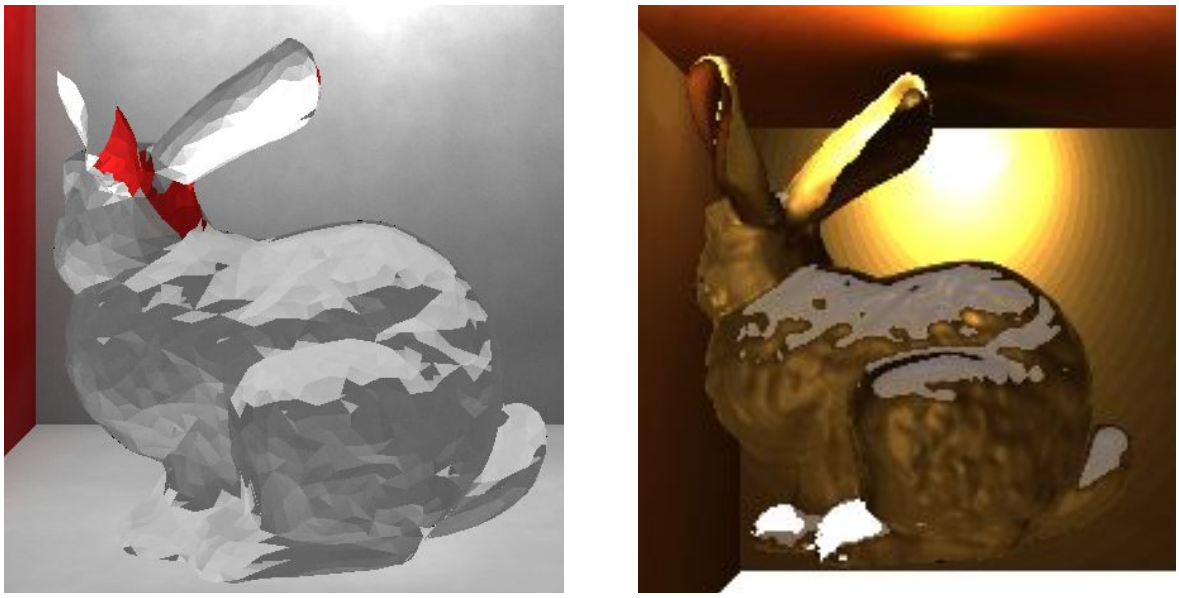
Physically based rendering is not the focus of my project, so I did not spend time developing my own implementation of BRDFs. Instead, I used the open source Mitsubishi Electric Research Laboratories (MERL) implementation (Matusik et al. 2003), which comes along with a large library of physically measured materials. (Anon n.d.; Anon n.d.)

I added a single function to this code, which converts from a incoming/outgoing/normal vector representation into the theta, phi in/out representation used to search the MERL library:

```
/*
 * This is the only function which was written by me. The rest are copied from the MERL database.
 * https://people.csail.mit.edu/wojciech/BRDFDatabase/code/BRDFRead.cpp
 */
Vec3 our_getBDRF(Vec3 incomingAngle, Vec3 outgoingAngle, Vec3 surfaceNormal, double* brdf) {
	//Imagine a plane at the surface with the surface normal
	//Project incoming and outgoing vectors onto this plane and calculate the angle between them (Azumith angle)
	const float theta_i = acos(cosBetweenVectors(incomingAngle, surfaceNormal));
	const float theta_r = acos(cosBetweenVectors(outgoingAngle, surfaceNormal));

	/*
	Azumith angle is the angle between the incoming and outgoing vectors when projected onto the plane given by the normal.
	Assuming isotropic BRDFs, we only need this angle instead of two separate phi angles.
	We can set one of the phi angles the the azumith and the other to 0 in our lookup, as only the distance between them matters
	*/
	const float phi_i = acos(calculateCosAzimuth(incomingAngle, outgoingAngle, surfaceNormal));
	const float phi_r = 0;

	//Lookup in BRDF from https://www.merl.com/brdf/
	double r;
	double g;
	double b;
	lookup_brdf_val(brdf, theta_i, phi_i, theta_r, phi_r, r, g, b);

	return Vec3(r, g, b);
}
```

Phong Normal Interpolation

Triangle normals are interpolated using a technique called Phong Interpolation. TO get the normal at a point on the triangle, the point is first converted into Barycentric coordinates, which are weights for each vertex of the triangle. The normal is then calculated as the weighted sum of the vertex normals, using the Barycentric coordinates as weights. This results in a smoothed look, which is very important for materials such as glass and plastic. (van Overveld & Wyvill 1997)

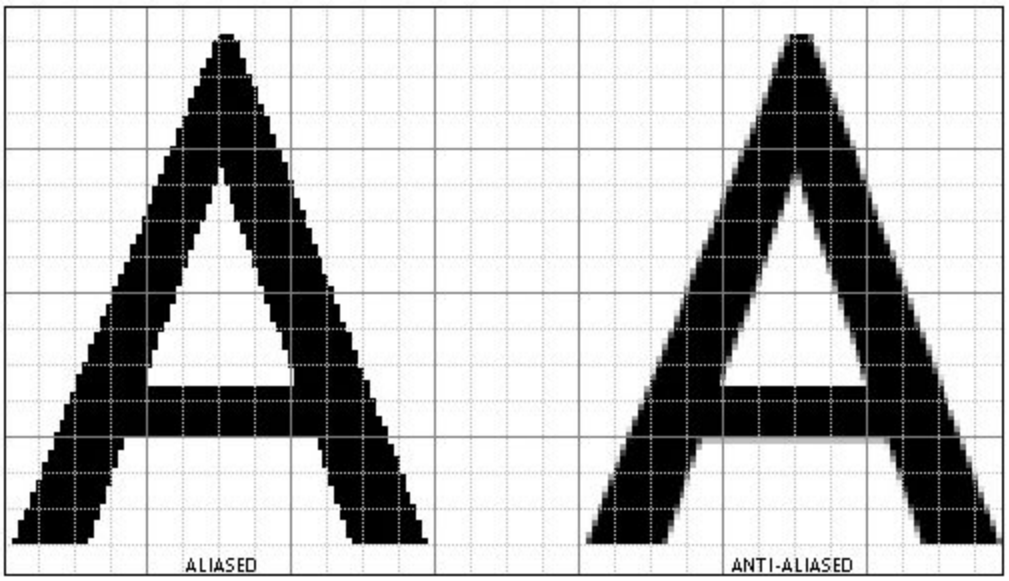


Before and after Phong Normal Interpolation

Supersampling

Supersampling is a technique for dealing with Aliasing artifacts, which are jagged edges caused when a continuous spatial surface is projected onto a discrete image plane. Supersampling renders the image at a resolution larger than the desired output resolution. This image is then sampled into the smaller output image using bilinear filtering, resulting in smoothened edges while preserving detail.

My project uses OpenCV's bilinear filtering function to resample the image after rendering at double the target resolution.



Reflection & Refraction

Materials can be specular, and specular materials have reflective and refractive probabilities. If a ray collides with a surface consisting of one of these materials, two rays are sent out. One is refracted, and one is reflected. The resultant colour of these two rays is multiplied by their probabilities and summed to get the specular colour at the intersection point:

```
Vec3 Scene::getTransmissiveComponent(IntersectionContext context, int
depth, int maxDepth)
{
	Vec3 refractiveComponent(0, 0, 0);
	Vec3 reflectiveComponent(0, 0, 0);

	if (context.material->reflectiveness > 0) {
		Ray reflectiveRay = Ray(*context.intersectionPoint,
reflectVector(-*context.inboundVector, *context.surfaceNormal));
		reflectiveRay.position += EPSILON * reflectiveRay.direction;
		reflectiveComponent = context.material->reflectiveness *
this->traceRay(&reflectiveRay, depth + 1, maxDepth);
	}

	if (context.material->refractiveness > 0) {
		Ray refractiveRay = Ray(*context.intersectionPoint,
refractVector(*context.inboundVector, *context.surfaceNormal,
context.material->indexOfRefraction));
		refractiveRay.position += EPSILON * refractiveRay.direction;
		refractiveComponent = context.material->refractiveness *
this->traceRay(&refractiveRay, depth + 1, maxDepth);
	}

	return reflectiveComponent + refractiveComponent;
}
```

Soft Shadows

Soft shadows are generated by sending multiple shadow rays from an intersection point to random points on an area light and averaging the results to get the shadow colour. Shadows should be filtered to reduce noise.

```
Vec3 Scene::getShadowComponent(IntersectionContext context, int
numberOfSamples)
{
	bool shadowed = false;
	int hitCount = 0;

	for (int i = 0; i < numberOfSamples; i++) {
		Ray shadowTest = Ray(*context.intersectionPoint,
-light->vectorTo(*context.intersectionPoint));
		shadowTest.position = shadowTest.position + (EPSILON *
shadowTest.direction);
		float shadowT;
		if (shadowTest.intersectsWith(*this, shadowT)) {
			if (shadowT <=
(light->getDistanceFrom(*context.intersectionPoint))) {
				hitCount++;
				shadowed = true;
			}
		}
	}

	return Vec3(0.5 * hitCount, 0.5*hitCount, 0.5*hitCount);
}
```

Code

Overview

The code was developed consistently over a month in C++, using Git for version control and a free BitBucket private repository to store the Git repo. Version control was required to enable destructive changes to code while maintaining the peace of mind of being able to revert to previous versions. The code was developed in a single branch with features and bug fixes being added in whole, working commits.

SourceTree was used to manage the Git repository, and Visual Studio 2017 Community was used to develop the code, due to its strong support for C++ in a windows environment.

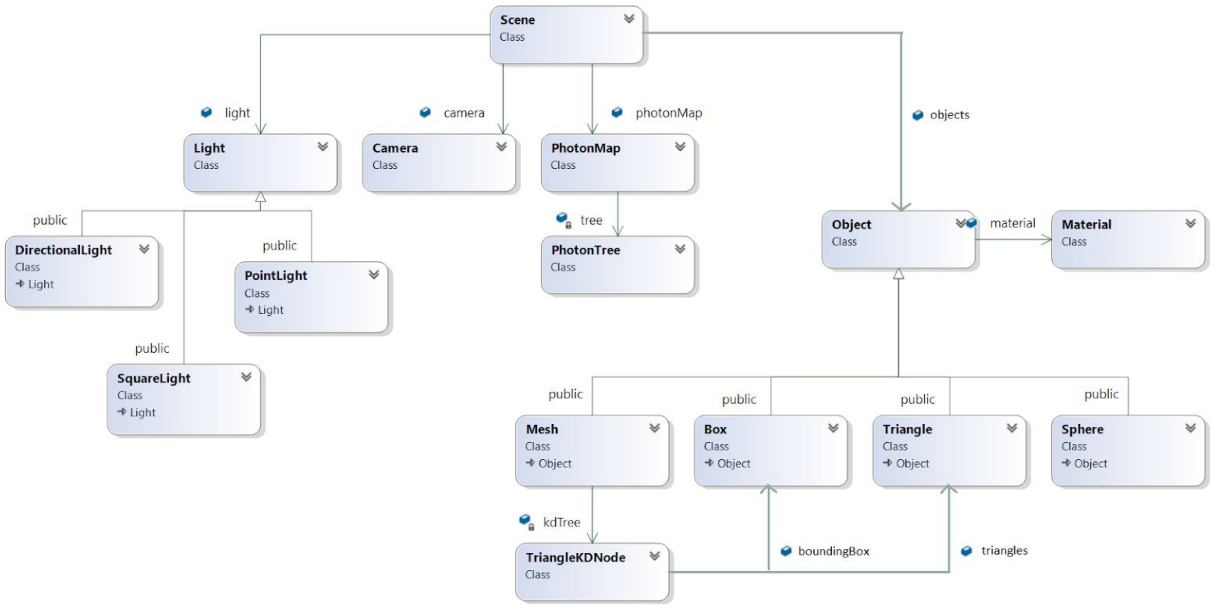
Design Decisions

Libraries

My solution utilises the following three libraries. These libraries were chosen to simplify the solution and prevent the unnecessary repeating of work that re-implementing the included features would entail.
Eigen - a performant linear algebra library, used in my solution for it's Matrix and Vector maths
LibIGL - a simple geometry processing library, used in my solution for it's OBJ file reading support, including support for vertex normals.
OpenCV - an open source computer vision library, used in my solution for its image display and export functionality.

Class Structure

I designed the class structure to be as extendable as possible. Any subclass of Object will be renderable, as will any subclass of Light. Here is the class diagram.
Given more time, I would have developed a generic KD-Tree class with customisable node and comparator objects, rather than the separate classes PhotonTree and TriangleKDNode. However, these were implemented differently. With PhotonTree optimised for memory consumption when mapping millions of photons.



Encountered Problems and Solutions

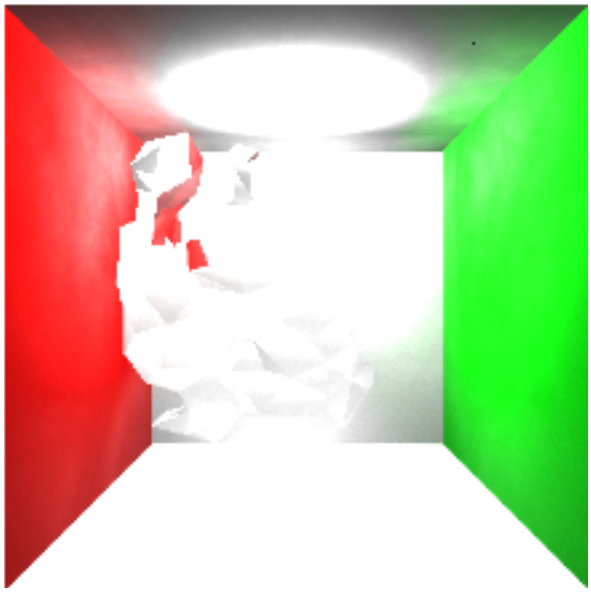
Slow renders

The naive approach to ray traced rendering (i.e. no threading and inappropriate data structures) results in extremely long render times for high triangle scenes. Due to the nature of the project, fast iteration times were required. The solution to this was to invest time in optimisation from the start of the project. Photon mapping was implemented originally with a KD-tree. Threading was implemented to speed up renders almost 700% on an 8 thread CPU. Meshes were implemented with simple bounding boxes and then later on with Hierarchical Bounding Volumes stored as KD Trees. Early on, a low poly bunny object was used for testing.

A GUI window also shows the render in its current form, updating every second. This allows for quick visual results to be discerned based on the first parts of the image without waiting for the full render to complete.

Over saturation

The RGB colour space for OpenCV's image objects ranges from 0-255 in each colour dimension. If the result of the pixel calculations comes out at over 255, overflow can occur, resulting in incorrect pixel values. This is solved by limiting the values to 255. However, this can result in a loss of detail, where large sections of the image are the same colour. This is resolved by multiplying the colours of each pixel by a constant 0<K<1. This constant is set manually and its value should be determined artistically for a render.



Overly saturated image. The brightness of the colours should be multiplied by 0<k<1 to reduce this

C++ linking issues

Working with C++ provides many issues, even to competent programmers. One of the most commonly recurring issues was circular includes. The C++ #include directive is naivley replaced by the compiler with the contents of the included file. This can result in situations where a child class can be defined before its parent class. This results in compilation errors. The solution is to forward declare the superclass when only a Pointer to it is required in another classes header, instead of including the superclass header. This removes the circular include path and fixes the issue.

Bibliography

Anon, MERL – Mitsubishi Electric Research Laboratories. Available at: https://www.merl.com/brdf/ [Accessed December 14, 2018a].

Anon, std::nth_element - cppreference.com. Available at: https://en.cppreference.com/w/cpp/algorithm/nth_element [Accessed December 14, 2018c].

Foley, T. & Sugerman, J., 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS '05*. Available at: http://dx.doi.org/10.1145/1071866.1071869.

Jensen, H.W., 2004. A practical guide to global illumination using ray tracing and photon mapping. In *Proceedings of the conference on SIGGRAPH 2004 course notes - GRAPH '04*. Available at: http://dx.doi.org/10.1145/1103900.1103920.

Jensen, H.W., 1996. Global Illumination using Photon Maps. In *Eurographics*. pp. 21–30.

Matusik, W. et al., 2003. A data-driven reflectance model. In *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*. Available at: http://dx.doi.org/10.1145/1201775.882343.

van Overveld, C.W.A.M. & Wyvill, B., 1997. Phong normal interpolation revisited. *ACM transactions on graphics*, 16(4), pp.397–419.

Popov, S. et al., 2007. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer graphics forum: journal of the European Association for Computer Graphics*, 26(3), pp.415–424.

Reinhard, E. & Jansen, F.W., 1997. Rendering large scenes using parallel ray tracing. *Parallel computing*, 23(7), pp.873–885.