NTNU
Det skapende universitet

Fotorealistiske bilder for objekt-gjennfinning

Arve Nygård

Master i datateknologi
Innlevert: juni 2016
Hovedveileder: Theoharis Theoharis, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Abstract

When developing computer vision algorithms that rely on machine learning, large image datasets for training and testing are required. Good datasets can be difficult to come by, and labour intensive to create. This thesis presents a framework for generating large, varied image datasets based on collections of 3D models. The framework is built around existing, open source tools. The thesis also presents an image dataset that was generated using this framework based on a well-known, labeled 3D model database, in combination with assets created by the author.

Dedicated to my daughter and fiancée

Table of Contents

List of Tables

List of Figures

List of Terms

depth of field	The region between the nearest and furthest points that are in focus.
GPU	Graphics Processing Unit.
Graphics Processing Unit	Also colloquially known as *graphics card*. A massively parallel data processor that is exceedingly fast at certain computations..
HDRI	High Dynamic Range Image.
High Dynamic Range Image	An image with high dynamic range.
material model	A mathematical model of how a group of materials interact with light e.g. metals or plastics.
Open Computing Language	A framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators..
OpenCL	Open Computing Language.
PLY	Stanford Triangle Format.
Scene Description Language	A language for describing a 3d scene: Light-, model-, material- and camera parameters..
SDL	Scene Description Language.
Stanford Triangle Format	A simple 3D model file format that is simple and easy to implement but that is general enough to be useful for a wide range of models.

Chapter 1

Introduction

When developing new algorithms for object recognition and retrieval, one needs an expansive dataset both for training and and testing the algorithm. In addition to having sufficient, the dataset should also be both complex varied enough to proivde a challenge to the algorithm in question. When a dataset has diversity in both the objects depicted, their surroundings they are located in and lighting conditions, the algorithm under test gets to prove that its generality. Finally each object should be displayed from multiple angles. If the dataset is too simlpe, an algorithm might do well on the test data, but fail to work in a real world setting. Good datasets of this kind can be hard to come by, as they are usually expensive to create. Having a photorealistic dataset adds to the weight of claims of usefulness - an algorithm that performs well on a dataset that closely resembles real photos will have a stronger claim to usefulness than an algorighm that performs well on an distinctly artificial dataset.

1.1 Motivation and Problem Description

Ideally, a dataset would consist of photographs of real objecs. However, this is obviously very labour intensive for something of this scope. Additionally, after obtaining the images, each one must to be manually labeled with their content. Generating images computationally has traditionally been expensive as well, due to the computation costs involved. However, with wide availability of cheap Graphics Processing Unit (GPU) cores, and software solutions that allow realistic rendering to be performed on the GPU, this has changed.

1.2 Goals

The goals of this thesis are:

1. *Provide a photorealistic image dataset based on a well-established labeled collection of 3D models.* The image dataset will be based on a well-established model

dataset in order to make comparison of computer vision algorithms easier, as the parameters of the model library are well-known. This will solve the immediate problem discussed above, where few such datasets are publicly available.

2. *Provide a framework to efficiently generate photorealistic image datasets from arbitrary model collections.* The framework will be based on an existing open source rendering application to take advantage of future improvements. The framework will aim to make it easy to create expansive image datasets based on any collection of 3D models, while being as customizable as possible. This solves another aspect of the problem mentioned above: Few good image datasets are publicly available, and those that are might not be suitable to specific computer vision applications. The framework should make it easy to create specific datasets when needed.

1.3 Contributions

The contribution of this thesis is two-fold:

- First, a large image dataset of over 25 000 labeled images capturing various objects from multiple angles. The images are based on the Princeton Shape Benchmark (PSB) which was published by Shilane et al. (2004) in 2004.
- Second, a framework for generating photorealistic image datasets from arbitrary collections of 3D models. The framework is based on an open source, physically based path tracing engine that runs computations on the GPU. The framework combines separate elements into a 3D scene to be rendered. This allows separate customization of materials, lighting conditions, and surrounding environment in order to tailor the parameters of the desired dataset to the user's wishes, as well as to limit the amount of work required.

1.4 Outline

This thesis is structured in the following manner:

- **Chapter 2 — Background** introduces the background material that is used in the rest of the thesis. First, existing, well-established datasets that are related to the topic of this thesis are listed. Next, an introduction to macine learning and some prominent computer vision algorithms are mentioned. This should give the reader a foundation to evaluate the different aspects of a good image dataset. Then, the theory behind photorealistic rendering is summarised: An introduction to the the basic elements that comprise a 3D scene, such as 3D models, materials and lights is given, followed by a description of the path tracing algorithim.
- **Chapter 3 — A large dataset for computer vision** introduces the dataset that is published with this thesis. The chapter continues with descriptions of the various dimensions of variance added to the images, explaining how each contributes to difficulty and generality of the dataset. Next, a detailed description of the camera setup

is given. Finally, the classification system from the Princeton Shape Benchmark is described. This chapter's main purpose is to highlight the features of the dataset from the perspective of someone who will use it to train and test a computer vision algorithm.

- **Chapter 4 — A framework for generating photorealistic image datasets — Part 1: Technical** introduces the framework published with this thesis. This chapter describes the technical aspects of the framework. The chapter starts by describing the specific rendering engine the framework is built around. Next, each major building block is described in detail, along with some of intents behind the design decisions that were taken regarding each part. This chapter aims to give a solid understanding of the framework in order for others to extend and improve the framework.
- **Chapter 5 — A framework for generating photorealistic image datasets — Part 2: Usage** explains how to use the framework in order to generate new, custom datasets or modify and improve the default dataset introduced in Chapter 3. The chapter contains instructions on how to compile the framework and prepare the models from the Princeton Shape Benchmark, as well as how to run the framework in order to start the image generation process.
- **Chapter 6 — Results** Discusses the qualitative properties of the generated images such as noise levels and image artifacts. The chapter then discusses the technical difficulties encountered during the generation process.
- **Chapter 7 — Conclusions and Further Work** summarizes the findings in this report, and suggests future work.

Additionally, the thesis consists of an *electronic attachment*. This attachment contains source code for the framework developed in the thesis work.

Chapter 2

Background

2.1 Datasets

2.1.1 Princeton Shape Benchmark

The Princeton Shape Benchmark (Shilane et al., 2004) is a database of approximately 1800 labeled 3D models representing a wide variety of objects such as animals, humanoids, airplanes and chairs. In addition a set of models, the benchmark also contains a set of classification hierarchies, that map similar models into groups. This database is what we will use as a basis for generating images.

2.2 Object recognition

The goal of an object recognition system is to recognize different types of real world objects in images, despite varying lighting conditions, camera angles, scales and so on.

Usually some form of Machine Learning algorithm is used to achieve object recognition: The algorighm is trained on a large training dataset in which images are labeled with their contents. After the training process is complete, the algorightm will hopefully have learned what distinguishes the different objects, and how to recognize each of them. Usually, a second part of te dataset is dedicated to testing: After training, the algorighm can try to classify images from the testing dataset, and we can measure how well it does.

A key part of object recognition is segmentation: The algorithm must be able to distinguish which parts of the image depict the object in question, and which parts are part of the background. When artificially rendering images, we automatically get a ground truth for segmentation in the alpha channel of the image.

2.2.1 Descriptors

After isolating the image and obtaining a region of interest, an algorithm will typically reduce the image region using a descriptor. A descriptor can be thought of as a compact way of representing important features of an image. A good descriptor would have similar values for images containing objects with similar features.

2.3 Rendering

In the context of this computer graphics, *rendering* is the process of turning a 3D scene into a 2D image. A scene may contain multiple 3D models, material definitions for surfaces, a camera, lights, and an environment definition. The scene is used as input to a *renderer*, which attempts to solve the Rendering equation (Kajiya, 1986) for the given scene in order to produce a photorealistic image.

The process combines the geometric surface information from the model with the material definitions and lighting setup to calculate how light bounces around in the scene and hits the virtual camera sensor.

2.3.1 3D models

A 3D model represents a physical body using a collection of points in 3D space (vertices), along with edges connecting these vertices. Together, these primitives form a set of connected polygons which describe the surface of the object.

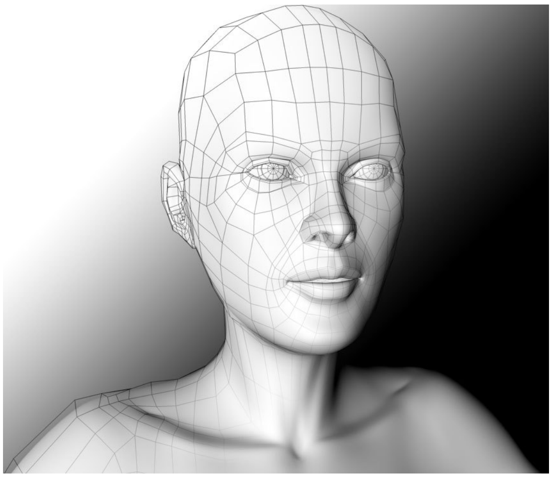

Figure 2.1: A polygon 3D model. Credit: Wikimedia Commons

3D models are usually created by an artist, however algorithmic reconstruction is also common. Models can be reconstructed from images, however they most often originate from specialized scanning equipment such as a 3D scanner. Models vary in size, ranging from around 200 vertices for a very simple model, to several thousands for a typical game model, to tens of millions for a 3D scanned model.

There are a plethora of 3D model file formats, with different encodings and feature sets. Formats are optimized for things like size, processing time, or simplicity. The model database we are using are in the *Object File Format* (OFF). This is a plaintext format, which makes it simple to parse.

2.3.2 Materials

In addition to geometric information, a model typically also has some material information attached to it. Not surprisingly, the material information defines what the related surface is composed of. Parameters like reflectance, diffuse color, transparency, index of refraction, and specularity affect how light behaves when it strikes the surface. There are many possible *material models* (also called lighting models), which typically vary between renderer implementations. A simplistic material model may only contain diffuse color information. For photorealism, a more advanced material model is required. Often, most of the previously mentioned parameters are tunable, as well as the Bidirectional reflectance distribution function (BRDF) (Nicodemus, 1965-07-01), and specialized parameters such as Sub-Surface Scattering (Krishnaswamy and Baranoski, 2004-09) for translucent materials like skin and wax.

For a modern renderer that targets photorealism, there are typically several specialized material models available, each suitable for describing a group of real-world materials. Often, you will find a separate material model for metals, one for dielectrics (glass), as well as others. This is both due to computational constraints, as well as to make it easier for the artist to author believable materials.

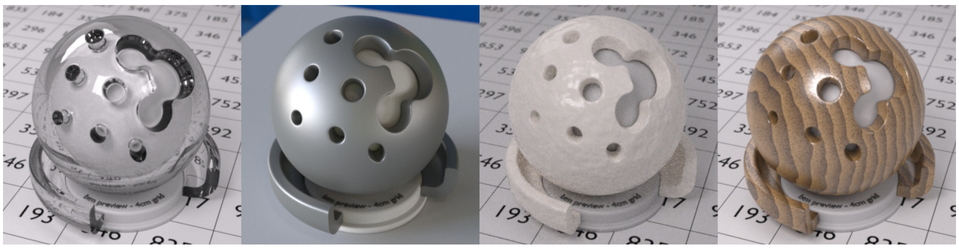

Figure 2.2: From left: Glass, Steel, Paint, and Wood materials applied to the same model

It is rare for an object to be uniformly made of a single material. Often, there are variations in color, reflectance, etc. along the surface of an object. As an example, imagine a piece of wood that is partly soaked in water. There will be color variations due to the wood grain, as well as variations in how light interacts with the dry versus the wet parts of the wood. To account for this, the surface elements of a 3D model can be mapped onto a plane (typically called the *UV plane*) in order to parameterize locations on the surface of the model. This allows the artist to vary material parameters across the model, often in the form of bitmaps that control the value of a given parameter across the surface.

Finally, not all materials are based on two-dimensional maps painted by an artist. Since the geometry is in 3D space, any 3D function can be used as input for material parameters such as color. 3D material funcitons is common for things like wood, or marble, where grain patterns should not follow the surface of the object. Another advantage of procedural materials like this is that they are not specifically tailored to any model, and can be used on any geometry, as opposed to materials that depend on a mapping of the surface to a plane.

2.3.3 Lighting

The renderer draws the image by calculating how light flows through the scene. There are several types of possible light sources: First, we discuss regular light entities. There are a couple types of light entities: point-, directional-, area- and spotlights. Point lights radiate equally in all directions, from a single point in space. Directional light has no origin, just a direction. Directional lights are usually used to emulate the sun. Area lights radiate out from a plane. Finally, spotlights radiate light from a point, but only at a given angle. Lights can have parameters such as Luminosity, color and attenuation. These lightsources are only an approximation of the real world: A point light source is physically impossible. A second type of light source is a 3D model. A model can be assigned a light-emitting material.

2.3.4 Image based lighting

In addition to in-scene lights, there is image based lighting (IBL). In IBL, a high dynamic range image (HDRI) projected around the scene, at infinite distance. The renderer then uses the image as a light source, where the values of the image at each direction describes how much light is arriving from that direction. A HDRI is often created by photographing a real world scene at multiple exposures and combining these images. This effectively encodes the lighting conditions of that particular scene for re-use in 3D scenes.

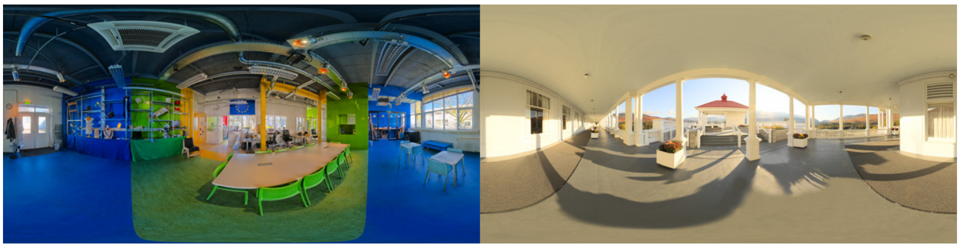

Figure 2.3: HDR Environment maps suitable of image based lighting. Credit: Bob Groothuis (`http://www.bobgroothuis.com/`)

There are several advanages of IBL. First, one is guaranteed a realistic lighting setup, since the image is sourced from a physical location with a regular camera. Secondly, the HDRI can be used as a background image for the scene, and one is guaranteed that the scene lighting perfectly matches any objects in the background.

It is possible to use a combination of IBL and regular lights in a scene.

2.3.5 Rendering equation

The Rendering equation describes the interaction of light between surfaces in a scene: At each particular position ($\mathbf{x}$) and direction (ω_o), the outgoing light (L_o) is the sum of the emitted light (L_e) and reflected light. The reflected light is the sum from all directions ($\int_\Omega$) of the incoming light (L_i) multiplied by the surface reflection (f_r) and cosine of the incident angle ($\omega_\mathrm{i} \cdot \mathbf{n}$). The equation also takes into consideration time ($\mathbf{t}$) and wavelength(λ). A common form of the equation is:

$$L_\mathrm{o}(\mathbf{x},\,\omega_\mathrm{o},\,\lambda,\,t) \,=\, L_e(\mathbf{x},\,\omega_\mathrm{o},\,\lambda,\,t) \,+\, \int_\Omega f_r(\mathbf{x},\,\omega_\mathrm{i},\,\omega_\mathrm{o},\,\lambda,\,t)\, L_\mathrm{i}(\mathbf{x},\,\omega_\mathrm{i},\,\lambda,\,t)\, (\omega_\mathrm{i}\,\cdot\,\mathbf{n})\,\,\mathrm{d}\,\omega_\mathrm{i}$$

Note that a solution to the rendering equation is the function L_o. The function L_i is related to L_o via a ray-tracing operation: The incoming radiance from some direction at one point is the outgoing radiance at some other point in the opposite direction. This means that there is a recursive element to rendering, and that the exact solution to the equation is approximated as we take limit of this recursion.

2.3.6 Rendering methods

There are several approaches to solving the rendering equation. One approach is based on finite elements methods, leading to the radiosity algortihm, where only light paths which leave a light source and are reflected *diffusely* some number of times (possibly zero) before hitting the eye are accounted for. [radiosity example.]

Another approach using stochastic methods (Monte Carlo estimation) has led to algorithms such as path tracing (Kajiya, 1986), photon mapping (Jensen, 2001) and Metropolis light transport (Veach and Guibas, 1997). We will focus on path tracing as this is the technique we will be making use of in this project.

In path tracing, a ray is traced through every pixel of the camera, into the scene. Once the ray hits a surface, a new ray is traced in a random direction of the surfaces hemisphere. This is repeated recursively until some max depth is reached. At each step of the recusion, the sum of emitted light and reflected light (returned from the next ray) is returned. This way, light is transported back up the chain to the originating pixel. This process is repeated multiple times per pixel and the resulting color is averaged. Due to the random nature of the reflected ray's direction, this method approximates the integral over the hemisphere at each reflecting point as the number of samples approaches infinity.

A simplified version of this algorighm follows.

```
Color TracePath(Ray r, depth) {
   if (depth == MaxDepth) {
      return Black;  // Bounced enough times.
   }

   r.FindNearestObject();
   if (r.hitSomething == false) {
      return Black;  // Nothing was hit.
   }

   Material m = r.thingHit->material;
   Color emittance = m.emittance;

   // Pick a random direction from here and keep going.
   Ray newRay;
   newRay.origin = r.pointWhereObjWasHit;
   newRay.direction = RandomUnitVectorInHemisphereOf(r.
    normalWhereObjWasHit);  // This is NOT a cosine-weighted distribution!

    // Compute the BRDF for this ray (assuming Lambertian reflection)
    float cos_theta = DotProduct(newRay.direction , r.normalWhereObjWasHit);
    Color BRDF = 2 * m.reflectance * cos_theta;
    Color reflected = TracePath(newRay, depth + 1);

    // Apply the Rendering Equation here.
    return emittance + (BRDF * reflected);
}
```

2.3.7 GPU rendering

Recent developments in graphics processing unit (GPU) processing, such as the *OpenCL* framework (Munshi and others, 2009) has enabled efficient implementations of path tracing renderers that run on the user's GPU. This has yielded a speedup of around 10x compared to CPU based renderers, due to the parallel nature of the computations required for path tracing.

2.3.8 Dynamic range

A regular computer monitor, as well as most standard image formats represent the color of each pixel using 24 bits - 8 bits for each of red, green and blue. This means that an image can only represent a limited number of brightness values. During the rendering process, the engine simulates how lights flow through the scene, eventually hitting the camera sensor. Since the scene can contain arbitrarily strong light sources, the range of values across different pixels on the sensors may vary by many orders of magnitude. The simulation is carried out using many more bits than the 8 in the resulting image. Since the simulation may contain a much larger dynamic range than is possible to represent in a regular image, there is an intermediate step before outputting the result: Tone mapping. The process of tone mapping consists of compressing the high dynamic range of the simulation result into the low dynamic range of the image. Tone mapping can be done manually, or automatically. When done automatically, an algorithm decides how to map the values between the two ranges. The result of automatic tone mapping is often that images with localized, very bright spots are normalized so that the surroundings are darkened. This effect is similar to auto-exposure settings often found on video cameras.

Chapter 3

A large dataset for computer vision

3.1 Overview

This chapter introduces the dataset that was generated as part of this thesis. We discuss some of the choices that were made regarding parameter selection and sources of variance, and how they can affect the training and testing phases for an object retrieval algorithm.

3.2 Dataset

Published along this thesis is a collection of 25382 images depicting 1813 different objects captured from 14 different camera angles. The images contain objects from the Princeton Shape Benchmark (Shilane et al., 2004) rendered with the physically based path tracing engine Luxrender. Each image is accompanied with a black and white binary mask that can serve as a ground truth for segmentation. This makes the dataset suitable not only for object retrieval and recognition, but also for segmentation benchmarking. To increase variety and get an optional difficulty gradient, as well as to combat overfitting, the objects are placed in different environments with varying charactersitics, and have different materials applied to them.

The models depicted in this dataset represent objects that in real life are of very different sizes - from insects to whales and airplanes. To handle the challenge of automatically generating photorealistic images of such a varied collection of models, which additionaly do not have any materal information linked to it, every model has been scaled down to a fixed size, and rendered as if they were miniature figurines of the given shape. This allows for a beliavable image while using procedural textures and camera placement.

3.2.1 Environments

The first environment resembles a typical studio scene. It has a smooth, diffuse background with very few features, as well as differently tinted spotlights. This should be the easiest environment with regards to segmentation, as there are very few distracting features. Additionally, the differently colored lights can potentially be used to aid with orientation analysis.

The second environment mimics a glossy table in an office setting. It uses a high contrast High Dynamic Range Image (HDRI) as the light source. Using a HDRI environment yields a more photorealistic result, as the scene's lighting is captured from a physical location. Additionally, we gain increased background complexity without the added computational overhead. This increases segmentation difficulty compared to the previous scene. Finally, since the environment map we are using is high contrast i.e. it has a very uneven brightness distribution, we get different exposure levels depending on camera angle. This is another challenge for the algorithm to overcome.

The final environment that was created for this dataset attempts to look like an outdoor platform with structural columns around its perimeter. The lighting setup here uses a procedural sun and sky light source, and has relatively sharp shadows. The main objective of this scene is to introduce obstructions, to further increase difficulty in recognizing the object in question.

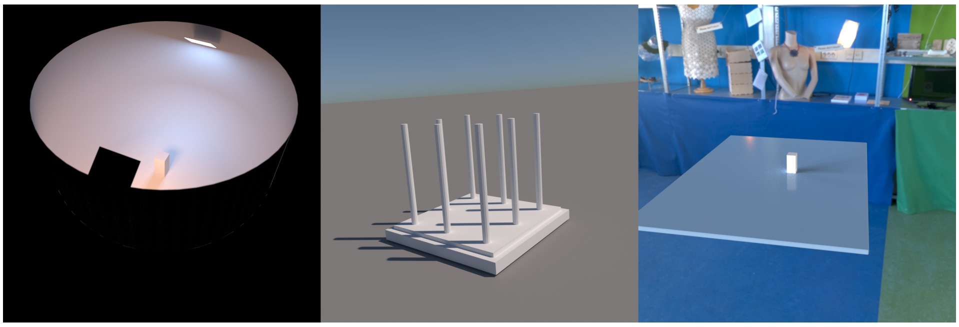
Figure 3.1: The three environments distributed with the framework.

3.2.2 Materials

In addition to the environment, each object is randomly assigned a material. When designing materials for a large render job like this, it is important to consider computational complexity. Different material models can have varying computational costs. In addition to being slower to evaluate, some materials also require many more samples before converging, due to the stochastic nature of the path tracing algorithm. In particular, transparent, refracting materials such as glass tend to require much longer render times to obtain acceptable noise levels, while simple, diffuse materials such as ceramics require less work.

Taking these points into consideration, while still attempting to provide some interesting variety, a small set of materials was written for this dataset. Keeping in spirit with the principle of graduating difficulty, the first material is a shiny plastic surface with a uniform, bright red color. This should be the easiest material w.r.t. segmentation, as the interior of the object has constant hue.

Next, we have a similar plastic material, except the plastic's diffuse color is a procedural marbled pattern generated by combining 3D noise functions. The goal is to get more internal detail on the object's surface.

The last material resembles a rough aluminium surface. It's based on a different material model than the previous two. This material will result in blurred reflections of the scene, allowing for recovering a lot of shape information, given an advanced enough analysis. However, it can be difficult to segment due to containing the same colors as its surroundings.

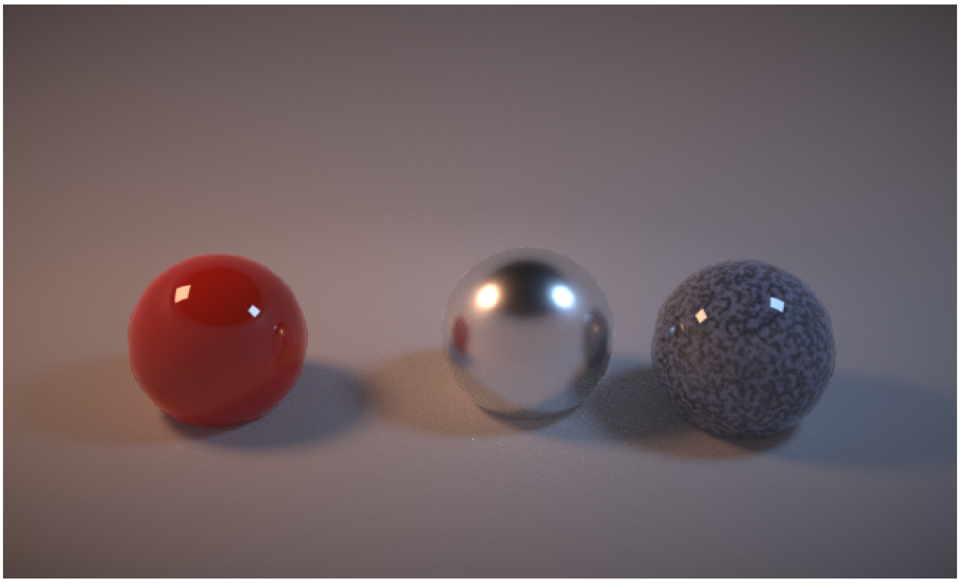

Figure 3.2: The three materials distributed with the framework

3.2.3 Camera

As mentioned above, each object is rendered from 14 different angles. In order to get good shape coverage, the camera angles are spread uniformly around a hemisphere above the model. To generate the camera locations, we started with a regular dodecahedron which bottom half was truncated: Any vertex lying below the Y=0 plane was removed from the set. The vertices vertex lying exactly on the Y=0 plane were kept. Additionally, two new vertices were introduced where two edges cross the Y=0 plane, to get better profile coverage. Table 3.1 lists each camera position relative to the object center.

The user is free to take advantage of this information and treat the dataset as registered images with known camera locations, or to ignore the camera location information and simulate a more natural scenario in which the object might be captured by a hand-held camera without precise knowledge of its location in relation to the object.

Since each object is on the order of centimeters in size, and the camera is close to the object, the camera has a somewhat shallow depth of field. This results in parts of the

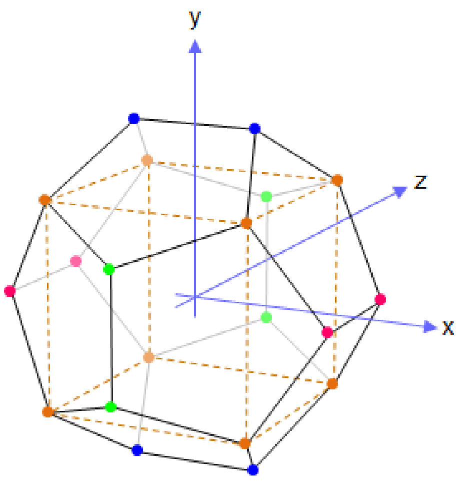

Figure 3.3: Camera positions correspond to the vertices on the top-half of a dodecahedron. Credit: Wikimedia Commons

Image no.	X	Y	Z
0	1.618	0.0	0.0
1	0.618	1.618	0.0
2	-0.618	1.618	0.0
3	-1.618	0.0	0.0
4	-0.618	-1.618	0.0
5	0.618	-1.618	0.0
6	1.618	0.0	0.618
7	1.0	1.0	1.0
8	-1.0	1.0	1.0
9	-1.618	0.0	0.618
10	-1.0	-1.0	1.0
11	1.0	-1.0	1.0
12	0.0	0.618	1.618
13	0.0	-0.618	1.618

Table 3.1: Camera positions relative to object center

model being out of focus, if they are too near or too far from the focus plane. The camera is always focused at origo. This is a side-effect of lens construction in physical cameras, and is usually more prominent on close-up photos.

3.3 Classification

In order to simplify evaluation of classification and retrieval algorithms, the Princeton Shape Benchmark includes a mechanism to specify groups of models into different classes or categories. Since our image dataset is based on these models, the class definitions applies to it as well. Images in our dataset are stored in folders named after filename of the model they depict. Shilane et al. (2004) created a simple plaintext file format to describe which models belong to which classes. The format contains a list of classes, and for each class, a list of models which belong to that class. Morevoer, a class can contain multiple subclasses. The result is that the models are assigned to a class hierarchy, giving the user control of the granularity they run their classifier against. The classification files are distributed together with the Princeton Shape Benchmark.

3.3.1 Classification files

With the benchmark, Shilane et al. (2004) provide a set of classification files, which contain categories of varying coarseness. Each classification splits the dataset in two equally sized groups, the training set and the test set:

- The base classification spans a large variety of classes such as animals, plants, airplanes, furniture, and vehicles. It has a multi-level hierarchy. The first levels are based on the gross function of the model, while lower levels further divide the parent levels into finer categories. For the training set, the base classification consists of 90 “leaf categories”, i.e. categories that models are assigned directly to. Similar categories are grouped together under “parent categories”. In total this classification contains 129 categories. For the test set the classification has 92 leaf categories and 131 categories in total.
- The second classification file is coarser: It contains 42 leaf categories out of 49 total for the training set, and 38 leaf categories out of 44 for the testing set.
- The third classification only contains 7 leaf categories and no parent categories.
- Finally, the coarsest level contains just two leaf categories for both sets: “natural” and “manmade”.

These classification files are distributed along with the image dataset published with this thesis. An example classification file is included in (B).

Chapter 4

A framework for generating photorealistic image datasets — Part 1: Technical

4.1 Overview

This chapter describes the framework that was used to generate the images in the dataset introduced in the previous chapter. We describe the framework's constituent parts, and how the parts work together to form an automated image generation process. We then present the content used to generate the default dataset in section 4.9. As we cover each part of the system, we discuss the technical decisions and considerations that were made.

4.1.1 Building blocks

The framework consists of six main components that work together in order to automatically generate photorealistic images suitable for training and testing object recognition and retrieval algorithms:

- First, we have the Luxrender *rendering engine*. The Luxrender engine is built as a library in order for our program to take advantage of the renderer's functionality. This allows our program to take control of the rendering process and enables automatic combination of resources in a way that is not supported "out of the box" by the Luxrender executable.

- Second, the *object database*: This is the collection of 3D models to be used as subjects in the images. This can be any collection of models. The only requirement is that the models can be converted to the PLY format described in 4.8.1, so that the system can parse them properly. Distributed along with the source code of the framework is the Princeton Shape Benchmark (see 2.1.1), which includes 1814 different models.

- Third, the *material database*. Each model is randomly assigned a material from the material database. Materials are defined in Scene Description Language (SDL) files. An overview of the SDL format is given in 4.8.2. A small set of materials are distributed as part of the framework.
- Next is the *environment database*. The program cycles to a new environment for each model. The environment is a complete scene, except for the subject model. It may consist of 3d models for the surroundings, lights, environemnt maps, and materials. An environment is typically described with an SDL file and a collection of models.
- We then have the *objectrenderer* program. This is the main program that runs the rendering process. It takes as input the material, environment, and object databases, and combines these into a scene for each object. It then renders the object from multiple angles and saves the resulting images in a folder corresponding to the current model. The program is linked to the Luxrender library and relies on it to do the rendering job.
- Finally, the framework includes some *utilities* to download and preprocess the models contained in the Princeton Shape Benchmark. This preprocessing stage is described in detail in Section 4.7. These tools may also prove useful for preprocessing different datasets, as they have been written to robustly handle a wide variety of models. They can be inspected and serve as examples should further preprocessing be required on other datasets.

The following sections discuss each component in further detail.

4.2 Rendering Engine

Luxrender (Grimaldi et al., 2015) is a modern, open source renderer which contains modules for doing photorealistic path tracing on the GPU. The core functionality is exposed as a library, and we take advantage of this to build a customized rendering program that is specialized to our needs. Luxrender has a solid API, with bindings for both C and Pyhton, making it amenable to automated rendering. Additionally, the scene description format Luxrender uses is in plaintext, which makes it easy to combine scene elements in an automated fashion. The scene description format is modular - we can store materials, lights and models separately, and feed them to the engine at runtime. When running on the GPU, the engine translates material definitions into Open Computing Language (OpenCL) shaders, and uploads these along with scene resources to GPU memory.

4.3 Object Database

The object database may consist of any valid Stanford Triangle Format (PLY) models. Output images are stored in a folder with a name corresponding to the models name:

For a model called *human.ply*, the output images would be saved to *output/human.ply/[0-13].png*. This facilitates image labeling.

Though not required, it is recommended to use models with precomputed normals in order to a avoid faceted surface appearance. It is possible to use the *normalsply* tool (see 4.7.4) to automatically compute interpolated normals for every surface on the model. For reasons explained in 4.7, each model should be scaled to a bounding box of 0.16 units, and the bottom part of the model should lie at $Z = -0.0574$. The preprocessing utilities that come with this framework will automatically apply these transformations.

By default, the *Princeton Shape Benchmark* (Shilane et al., 2004) object database is distributed along with the framework.

4.4 Material Database

While the PLY format does support texture coordinates, vertex colors and similar, these are not used in this framework. Due to the sheer number of models in a typical scenario, manually authoring separate materials to match each model is not feasible. To facilitate using large model databases without authoring explicit materials for each model, the framework instead relies on procedural or parametric materials where the points on the model surface in world space act as positional parameters. This still allows for a wide variety of realistic materials, however each object is limited to a single material in this approach.

The framework comes with a small set of example materials: Aluminium, Red plastic and Marbled plastic . These materials were translated to SDL format from the Luxrender project's public material database, which is available at `http://www.luxrender.net/lrmdb2/en/`.

Luxrender's GPU engine currently supports 13 different material models, listed in 4.1. Each of these material models have different parameters, which are described in detail in the Luxcore SDL reference manual, available at `http://www.luxrender.net/wiki/LuxCore_SDL`. When creating new materials, it is important to consider the computational load the material requires. Some materials are much slower to simulate than others. This can heavily impact image quality when rendering with a fixed time limit for each image.

4.5 Environment Database

An environment definition consists of three parts: A set of PLY models for geometry, as well as materials and light sources defined in SDL.

The environment geometry typically consists of a platform or floor that the object sits on, in addition to other elements such as background walls or other elements intened to add detail or obstructions. There are a couple of key constraints that should be taken into consideration when creating geometry for an environment: First, *ground level*, i.e. the height at which each object is placed, is fixed at $Z = -0.0574$. This means that any pedestal or floor or other geometry intended for the model to sit on should be exactly at this height to avoid clipping or a floating object. Second, the scene must be consistent across all camera angles around the object. A typical *studio render* backdrop screen will

Material	Description
archglass	Architectural glass. Simplified material model, only used for windows.
carpaint	Advanced glossy material: diffuse surface with mutliple reflective coatings.
cloth	Textiles. Has presets for denim, silks, cotton, wool and polyester.
glass	Basic, smooth dielectric. Adjustable index of refraction
glossycoating	Used in combination with another “base” material.
matte	Simple, matte material. Very quick to render.
mattetranslucent	used for things like wax or translucent rubber.
metal2	Metal. Has presets for aluminium, silver, copper, gold, and amorphous carbon.
mirror	100% reflective material
mix	A blend between two different materials. Used to combine effects.
null	Not rendered
roughglass	Represents a frosted glass surface.
roughmatte	Similar to matte, but with microfacet simulation.
velvet	Represents a dark material with a colored, fuzzy surface.

Table 4.1: Material models supported by the Luxrender GPU engine

not work well, as the backdrop would only work for a limited set of camera angles. Unless using a HDRI map or procedural sky, care should be taken to cover any background regions with geometry. Otherwise, a 100% black background would end up in frame. Finally, it is crucial to consider the camera distance when creating environment geometry. The camera is positioned at points on a hemisphere around the object. The camera is always at distance of 0.252 units from origo. Any walls or background objects should be placed further away than this in order to avoid clipping and obstruction issues.

Any materials can be used in the scene, however it is wise to consider rendering time when authoring environments.

The scene also needs one or more light sources. This can either be parametric lights such as a *spot light*, a procedural *sun and sky*, or a HDRI map. Each of these types are represented in the example environment definitions included with the framework.

4.6 Main program

The *objectrenderer* program is the main program that drives the rendering process. It is responsible for combining elements from the environment-, material-, and object databases into complete scenes, and render them. The program loops over all the models in the object database, and renders each model sequentially.

The Luxrender engine supports a concept they call *dynamic scene editing*. Dynamic scene editing allows the client program to modify the active scene without having to transfer the entire new scene to the GPU. The library minimizes the amount of work required to update the scene and only transfers the modified parts of the scene to the GPU. Since the intended use case for this framework is to render thousands of different objects with a limited set of materials and environments, we take advantage of this in order to improve rendering time: To avoid building a completely new scene on every scene change, the program keeps track of any models added to the scene, and when it's time to render a new object in a new environment, the program instead deletes the previous object and environment models, and adds the new ones to the scene.

Another optimization the program employs is *material caching*. Because the program utilizes the GPU to render the images, the material definitions have to be translated into a format that the GPU understands. For complex materials this can be an expensive process. When the program starts, it loads all material definitions from the material database, as well as all materials defined in the environment database. These materials are then compiled into GPU shaders, and uploaded to graphics memory. Each material is kept in the scene - and thus in GPU memory for the duration of the job, which means they do not have to be reuploaded and recompiled everytime and object or environment changes. When new models are added to the scene, the program simply assigns materials that are already in memory to them.

4.6.1 Camera

The camera is placed along evenly distributed points on a hemisphere around the object, pointing toward the scene origo. These points are calculated from the vertices of a regular half-dodecahedron of unit size. Table 3.1 contains a list of the exact positions used as input. To get the physical scale right, the points are scaled by a distance factor to bring the camera closer to the object. The result is that the camera is always 25.2 centimeters away from the object center. Figure **??** illustrates camera placement.

Each object is scaled to fit within a bounding box of 16 centimeters, and is placed roughly 6 centimeters below the origo. This results in the center of the model being more or less at the center of frame. The camera has a field of view of 49 degrees, which makes the object fill most of the frame, while still leaving some space around it. The rendering engine simulates how light interacts with the camera lens. Due to the small size of the objects and the proximity of the camera, the depth of field is relatively shallow. In the current iteration of the framework, camera settings are hard-coded.

4.6.2 File Structure

When launching the program, the user has to specify a configuration file as an argument. This configuration file contains settings for the rendering engine. The default configuration file is located in a subfolder called `scenes/base.cfg`. Additionally, the program looks for materials and environments in paths relative to the configuration file:

- The environment database is located at `scenes/environment/`. Since an environment definition consists of multiple files, each environment has it's own subfolder.
- The materials database is located at `scenes/materials/`. Each material is fully defined in a single file. The filename must match the material's name.
- The model database should be in a sibling directory next to the `scenes/` directory, called `models/`. Each PLY model sits directly in this directory.

This file structure is already populated with the example resources that were used to generate the dataset in Chapter 3.

4.7 Model Preprocessing Utilities

Since we are using an automated process with multiple camera angles, it is important for each subject model to occupy a predictable segment of space in order to guarantee that the entire object is within view of the camera from every angle.

The framework includes an utility to automatically normalize any 3D models given as input. The utility is a short program based on PLY Tools by Turk (1998). This utility performs the steps described below. Some of these steps are tailored to the Princeton dataset we use in this thesis, and might need to be changed omitted when using different models.

The normalization process consists for four phases:

4.7.1 Orientation

Most PLY files come in a coordinate system where the Y axis is considered the "up" direction. LuxRender uses the Z axis as it's "up" direction. Therefore, it is necessary to rotate subject models such that they are oriented in a meaningful way. We swap the Z and Y axes by combining two 90° rotations - first around the X axis, then around the (new) Z axis.

4.7.2 Scale

The models are scaled uniformly along all three axes so that it fits within a bounding box of size 0.16 m. This ensures that the whole model fits in the camera's view, regardless of the original model's size.

4.7.3 Translation

We read the minimum Z value of the model's vertices and translate the model accordingly, so that it rests on the ground. Ground level is set to $Z = -0.0574$, which simplifies camera placement: A camera located at $Z = 0$ pointing towards origo would have the center of the model in the middle of the frame, given the model scale and camera parameters mentioned above.

We then translate the model along the X and Y axes to ensure that its *center of mass* is at the center of the scene. This last step is necessary in order to avoid issues with models that have a main body, with less significant attachments (for example a cable or a string) going off in some direction. Using the geometric center would push the main body off-center and cause unexpected results.

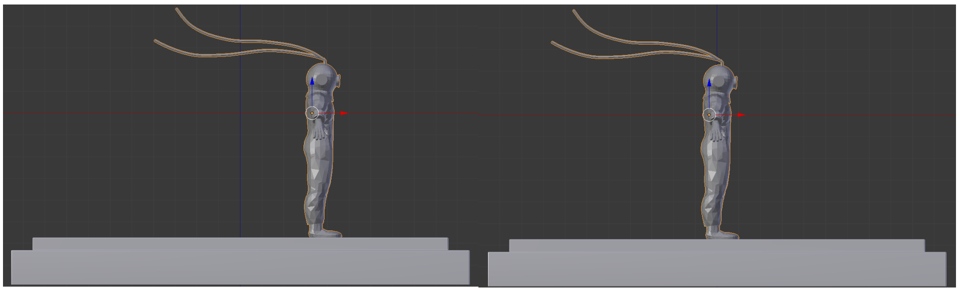

Figure 4.1: Geometric center vs. center of mass

4.7.4 Surface Normal Interpolation

Many models only contain the geometric information given by vertex positions and corresponding edges. However, while the PLY file format supports it, information about surface normals is often missing. When no normal information is present, the rendering engine will assume that each polygon is a flat surface, and the result is a faceted look in the final render. This looks decent on certain types of models, but models with organic shapes look very unnatural. To combat this, we calculate and store normals at each vertex, where the vertex normal is the average direction of its surrounding vertices. When the rendering engine reads a model with vertex normals attached, it interpolates this normal across the surface of a face, resulting in a smoother look. While this approach may have suboptimal results on low polygon models - and especially those with sharp edges and inorganic shapes, it is the best available alternative given a model without artist-defined normals such as the models in the Princeton Shape Benchmark. This step should be omitted for any user-provided models that already contain artist-created vertex normals.

The utility can be found at `bin/convert`. It's a shell script and can be inspected with any text viewer. The supporting binaries are located in the same folder.

4.8 File Formats

4.8.1 PLY - Stanford Triangle Format

The model file format Luxrender can read is called *Stanford Triangle Format* or *Polygon File Format* (PLY). This is a format for storing graphical objects that are described as a collection of polygons. The format is designed to be simple to parse, while still being general enough to represent a wide range of 3D models. The PLY specification contains both a binary and a plaintext version of the format.

A PLY file is built up of three parts: A Header, Vertex List, and a Face list. Additional properties such as vertex normals or vertex colors may follow the Face list. The header identifies the file as a PLY file, as well as how many vertices and faces the model contains, and how these vertices and faces are stored. If the model has any additional data stored, it is mentioned in the header as well. Following the header, each vertex and then face is listed in the format indicated in the header. Any optional properties follow after the face

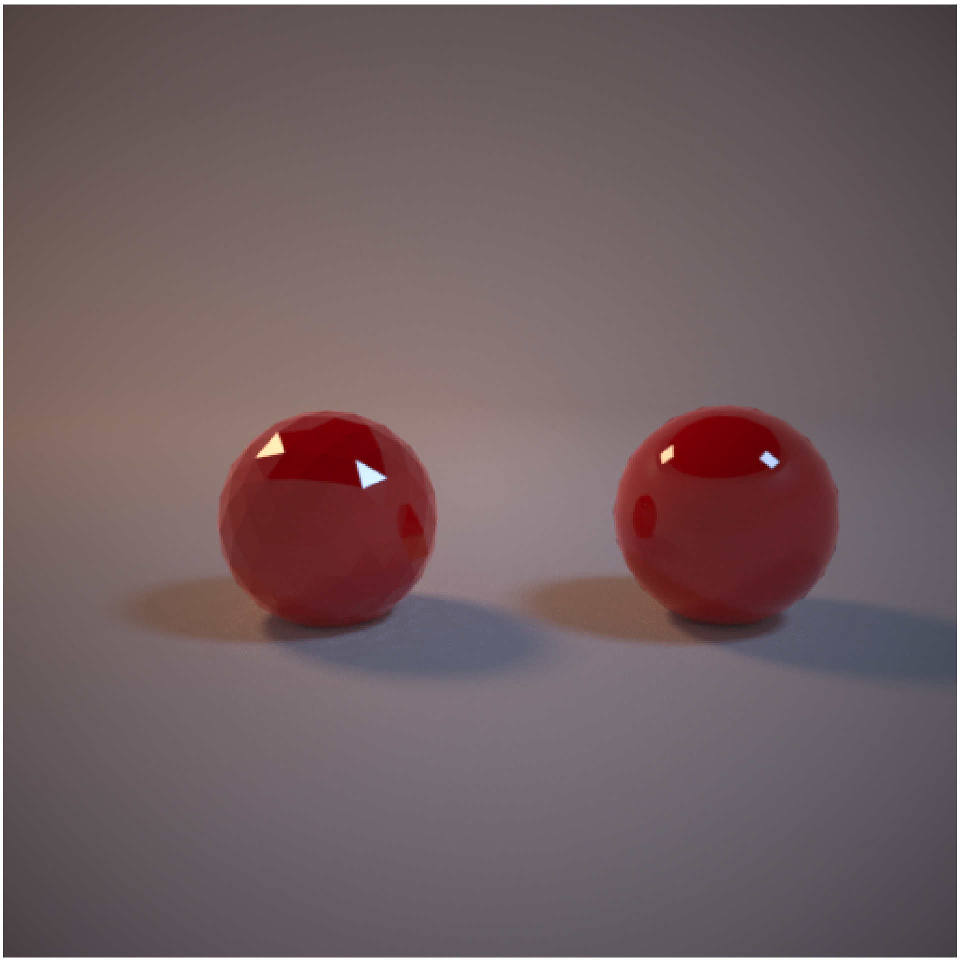

Figure 4.2: A 3D model with and without vertex normals

list. An example PLY file is shown in Listing A.1.

The models in the Princeton Shape Benchmark are *Object File Format* (OFF) files. The OFF format only differs from the PLY format in that it has a different header structure. The framework includes a simple script that convers OFF models into the PLY format.

4.8.2 SDL - Scene Description Language

The Scene Description Language (SDL) is a plaintext format that Luxrender can parse. In combination with any referenced PLY models, an SDL file should contain specifications for everything required to render a scene. In particular, the file must contain at least one light source, a camera, and should contain one or multiple models.

The basic building block of an SDL file is a *property*. A property is a key-value pair, where the "." character can be used in keys to conceptually group together different properties. An example material could look like Listing 4.1. Here, "scene.materials.shell" is the root of a node grouping together the ".type", ".kd", ".ks", etc. leafs. In addition to defining scenes, the SDL format is also used to configure the rendering engine.

```
        scene.materials.shell.type = glossy2
        scene.materials.shell.kd = 0.5 0.0 0.0
        scene.materials.shell.ks = 0.5 0.5 0.5
        scene.materials.shell.uroughness = 0.1
        scene.materials.shell.vroughness = 0.2

```

Listing 4.1: Material definition in SDL

4.9 Default resources

Included with the framework is a collection of resources that was used to generate the image dataset introduced in Chapter 3. These resources consists of 1814 3D models from the Princeton Shape Benchmark, three different environments, and three different materials, as well as a render engine configuration file.

While these resources allow a user to reproduce the dataset on their own, the framework is designed with the intent of allowing users to expand, modify and otherwise improve the resulting dataset. The author readily admits that his artistical skills are severely lacking, and there is a lot of headway to be made towards generating a truly photorealistic dataset.

4.10 Meta images

When rendering a scene, the rendering engine simulates how light rays interact with the surfaces of the scene after exiting the light sources, and obtains a final image by recording how much light hits each pixel on the cameras virtual sensor. To facilitate these simulations, the engine keeps track of a lot of data about the scene that is not used directly in the resulting image. Some of this information is stored in *screen space* (i.e. as seen from the cameras point of view). These screen-space buffers can be output as images, which we call *meta-images*. It is conceivable that several of these can be useful for certain classes of object-recognition algorithms. Image outputs are defined in the configuration file described in 4.6.2.

Below is a list of some of the most relevant output types for object recognition. The full list of supported output types is listed in Table C.1.

- `DEPTH`: Each pixel's value corresponds to the distance between the camera plane and the scene.

- `SHADING_NORMAL`: Each pixel is colored according to the direction of the surface normal at the corresponding point. This may be affected by materials if they contain *bump* or *normal maps*, or stored vertex normals. A similar output called

GEOMETRY_NORMAL ignores any normal-modifications and is only affected by the pure geometric shape, resulting in a more faceted output.

- OBJECT_ID: The profile of each object in view is filled with a distinct color.
- OBJECT_ID_MASK: Isolates a single object by ID. All pixels covered by this object is colored white, all other pixels are colored black.

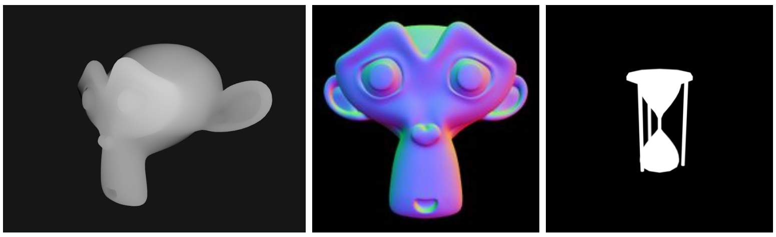
Figure 4.3: Meta images. From left: DEPTH, SHADING_NORMAL, OBJECT_ID_MASK

Due to space constraints, the author has decided not to enable all of these meta images, despite their potential usefulness. By default the framework only outputs regular images, intended for direct use with object recognition, as well as binary masks for each image, which can help with verifying segmentation.

Chapter 5

A framework for generating photorealistic image datasets — Part 2: Usage

5.1 Customizing the dataset

One of the guiding principles when designing this framework was that it should work with a wide variety of assets. The models and materials that are distributed with the framework are intended to serve as examples on how to build a photorealistic image dataset, but the author has tried to make the framework as general as possible, and he has tried to avoid specializing it towards the dataset used during development. The hope is that the framework will provide a means for other researchers to generate image datasets from different collections of models, perhaps with improved materials or environments, to cover other challenges in computer vision.

To this end, the author hopes that this thesis along with the provided with the framework provides a good foundation for generating other quality datasets. Due to the way the framework combines multiple elements into a scene, there are three main parameters that contribute to variance in the resulting image dataset: The Object database, Environment database, and Material database. The default assets contain very small environment and material databases compared to the size of the object database, so there is a lot of room for improvement here. Expanding these two would certainly help improve the generated dataset in that it would be more varied, and require algorithms that can handle more general cases.

When creating a custom dataset, the thechnical constraints described in Chapter 4 should be taken into consideration. The default assets should also serve as a good starting point and can be inspected in order to aid in creating new assets.

5.2 Building and running

5.2.1 Dependencies

Due to being built around the Luxrender library, the framework has a list of dependencies that must be present in order for it to compile. The framework was developed on and targets Ubuntu 15.10, but other Linux distributions should work as well, albeit some additional setup and configuration might be necessary. The following packages are required in order to compile the main program:

- CMake 3.2
- curl
- Boost 1.58
- OpenGL
- GLEW
- OpenEXR
- OpenImageIO
- libpng
- libtiff
- GCC 5

Some additional dependencies that are not available through Ubuntu's package repository are distributed with the framework. In addition to these packages, an OpenCL driver must be installed on the computer. In order for CMake to find the OpenCL headers, the user may need to export an environment variable corresponding to their driver. CMake looks for one of the following environment variables when trying to locate OpenCL: `ATISTREAMSDKROOT`, `AMDAPPSDKROOT`, `CUDA_PATH`, and `INTELOCLSDKROOT`. One of these should point to the location of the installed SDK.

5.2.2 Docker

The Docker website describes Docker as follows:

> Docker allows you to package an application with all of its dependencies into a standardized unit for software development.
>
> Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in.

Since the framework depends on multiple packages of specific versions in order to compile against the Luxrays libraries, the author has included a Dockerfile - a file that specifies to Docker how to build a container and run the program. This should make the process of building and running the program easier for users. When called, Docker will parse the file, download any requirements, build the framework and run the main program, without intervention from the user. However, by default the GPU is not exposed to programs running inside a Docker container.

At the time of writing, Docker does not support exposing the GPU to client programs. However, Nvidia has created a plugin called *docker-nvidia-plugin*, which enables access to Nvidia GPUs. The plugin is available at `https://github.com/NVIDIA/nvidia-docker`. Any users attempting to build the framework on a computer with a different brand of GPU, will have to do so manually. However, the Dockerfile can still serve as a reference on how to configure the system.

To build the docker image, simply issue the command docker build inside the framework directory.

5.2.3 Compiling

The framework is designed to be built and run with a few simple commands, given that all system dependencies are present. It comes with a Makefile that automates both the compiling stage, as well as downloading and pre-processing the models from the Princeton Shape Benchmark.

The command make convert is used to prepare the model database. First, it patches and compiles the *plytools* programs used by the model preprocessing utilities, then it downloads the Princeton Shape Benchmark. Finally, it extracts each model in the archive and converts it into the format expected by the main program.

The command make objectrenderer will call CMake in order to compile the *objectrenderer* program. It is important to make sure the correct environment variable for OpenCL has been set first, as described in 5.2.1. Otherwise the compilation will most likely fail.

5.2.4 Running the program

The framework is designed to do its job with as little human intervention as possible. After preparing the dataset and compiling the program, the program is simply invoked and left alone until it returns. Images are output as configured in the configuration file. The images of each object is stored in a separate folder named after the model's filename. The program only needs a single argument: The time it should spend rendering each image. To invoke the program, issue the following command: `./objectrenderer -render-time=<seconds>`, where `<seconds>` is replaced with a number.

Chapter 6

Results

The framework was employed to generate the dataset described in Chapter 3. In total, 25382 regular images were created, each accompanied with an object mask image. The images had a resolution of 512 by 512 pixels. The generation process was done on the author's personal computer with a 2013 AMD graphics card. Render-time per image was set to 15 seconds, and the whole job took roughly one week to finish. On average, each image was rendered at a rate of around 10 million samples per second, resulting in a sample count of between 400 and 600 per pixel in most images. Some combinations of materials and environments were slower to calculate than others. This caused the sample count to vary across scenes. Figure 6.1 Shows the image series generated for two objects, along with the mask for each image.

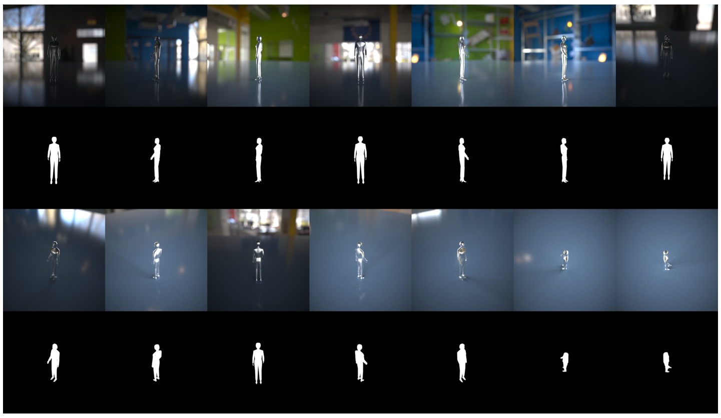

Figure 6.1: Image series for a human-type figure. Rendered image and image mask.

6.1 Noise

Since the path tracing process is a stochastic iterative approximation of a solution to rendering equation, the resulting images will always have some inherent noise. For each pixel in the image the scene is sampled multiple times, and the final pixel value takes the average value of all samples for that pixel. Naturally, as the number of samples goes up, the pixel variance across the image goes down. Experiments showed that noise levels were acceptable for our purposes at around 500 samples per pixel. Figure 6.2 illustrates how noise levels fall off as samples increase.

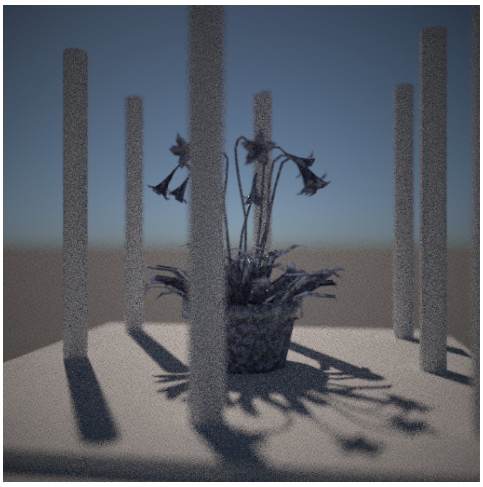

Figure 6.2: Noise resulting from too few samples. The rendering was stopped at 37 samples per pixel.

6.2 Fireflies

Certain scene configurations can result in overly bright spots scattered around the image. These spots are artifacts stemming from numerical instabilities in solving the rendering equation (2.3.5), and are often dubbed *fireflies*. Fireflies tend to appear in situations where light bounces off multiple reflective surfaces. Some image series on the dataset were affected by fireflies, as shown in Figure 6.3. As opposed to noise, fireflies can not be reduced by increasing the number of samples.

Luxrays supports a feature called outlier rejection, which can detect and reject a large portion of fireflies that occur in images. However, at the time of writing, this feature is only supported in the CPU renderer - the GPU code cannot do this currently. This might be changed in future updates.

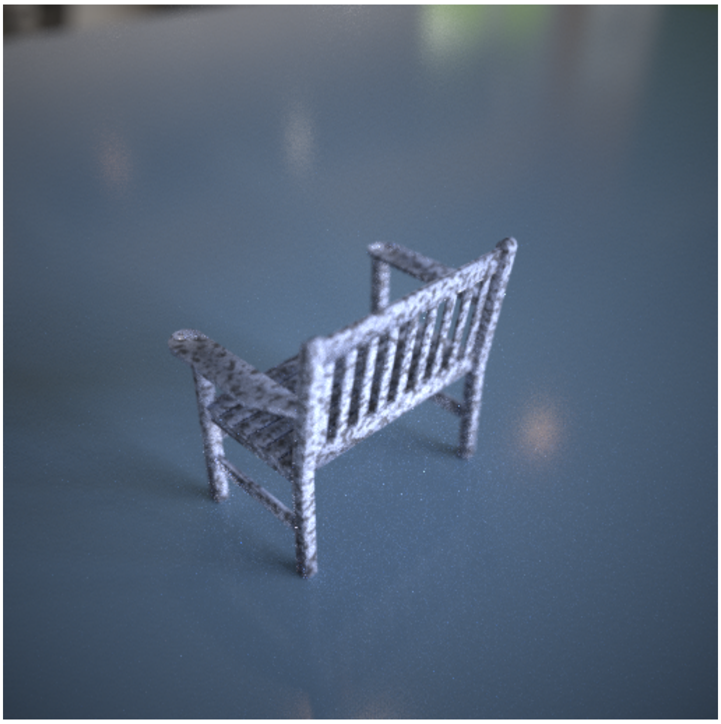

Figure 6.3: Firefly artifacts. Note the small, bright spots below the bench.

6.3 Technical issues

During the week-long render job, the program crashed two times with a *segmentation fault*. The source of the crash seemed to be stem from the Luxrender library: Debugging information indicated that the error occured in a thread that was communicating with the GPU at the time. Memory usage was monitored throughout the process and stayed constant.

Another issue with the Luxrender GPU engine that was encountered where mesh lights were sometimes not removed from the scene properly. A mesh light is a 3D surface that is assigned a special light-emitting material. If a scene contained only mesh lights and no other light sources, the lights would persist after clearing the scene elements, and thus appear when loading a new, different environment. This bug did not occur when using the CPU version of the rendering engine. Both these issues are likely to be fixed in future versions of the Luxrender software.

Chapter 7

Conclusions and Further Work

In this chapter, our findings are summarized and directions for future work are provided.

We have built a framework that can leverage assets in a combinatorial way to produce a dataset with a high degree of variance across several dimensions. In addition to the shape variance that is intrinsic to the model, features such as highlights and shadows, surface material and noisy or occluding surroundings can provide varying degrees of difficulty to a dataset. Using the framework to generate a full image dataset from the Princeton Shape Benchmark gave rise to some observations: Around 500 samples per pixel results in acceptable noise levels, and obtaining this degree of sampling is fast enough on modern hardware to be achievable and practical for large rendering jobs. There were some technical issues with the rendering library we built the framework around - however this is an open source project in active development, and these issues are likely to be remedied in the near future.

One of the main goals we set out to complete was to generate a dataset that could pass as photorealistic. This proved to be difficult, mainly due to two reasons: First, the 3D models used as a source were very low resolution. This resulted in lack of high frequency shape detail, as well as some render artifacts around the silhouette for many models. Second, the author is severely lacking in artistic ability. Creating believable environments proved to be a great challenge to the author. The difficulty was amplified because the scene had to look believable from virtually any angle, so many types of "camera tricks" could not be employed. The lack of shape resolution also strongly affected how the materials appeared. High frequency reflections that often appear in the real world were very rare and far between.

7.0.1 Further work

The author strongly encourages the improvement of existing, or creation of additional materials and environments for the framework. This would greatly improve the resulting look of generated images. Next, there are some technical improvements that can be done to the framework, chiefly adding the ability to read camera setup from a configuration file in order to enable another parameter to tune for the end user. Finally, employing the framework to render objects from a collection of models of higher quality than those in the Princeton Shape Benchmark would yield improved results.

Bibliography

Grimaldi, J.-P., , et al., 2015. Luxrender.
URL `http://www.luxrender.net`

Jensen, H. W., 2001. Realistic image synthesis using photon mapping. AK Peters, Ltd.

Kajiya, J. T., Aug. 1986. The rendering equation. SIGGRAPH Comput. Graph. 20 (4), 143–150.
URL `http://doi.acm.org/10.1145/15886.15902`

Krishnaswamy, A., Baranoski, G. V., 2004-09. A biophysically-based spectral model of light interaction with human skin. Computer Graphics Forum 23 (3), 331–340.
URL `http://doi.wiley.com/10.1111/j.1467-8659.2004.00764.x`

Munshi, A., others, 2009. The opencl specification. Khronos OpenCL Working Group 1, l1–15.
URL `https://www.khronos.org/opencl/`

Nicodemus, F. E., 1965-07-01. Directional reflectance and emissivity of an opaque surface. Applied Optics 4 (7), 767.
URL `https://www.osapublishing.org/abstract.cfm?URI=ao-4-7-767`

Shilane, P., Min, P., Kazhdan, M., Funkhouser, T., Jun. 2004. The Princeton shape benchmark. In: Shape Modeling International.

Turk, G., 1998. Ply tools.
URL `http://www.cc.gatech.edu/projects/large_models/ply.html`

Veach, E., Guibas, L. J., 1997. Metropolis light transport. In: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '97. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp. 65–76.
URL `http://dx.doi.org/10.1145/258734.258775`

Appendices

Appendix A

PLY file format

```
ply
format ascii 1.0           { ascii/binary, format version number }
comment made by Greg Turk  { comments keyword specified, like all lines }
comment this file is a cube
element vertex 8           { define "vertex" element, 8 of them in file }
property float x           { vertex contains float "x" coordinate }
property float y           { y coordinate is also a vertex property }
property float z           { z coordinate, too }
element face 6             { there are 6 "face" elements in the file }
property list uchar int vertex_index { "vertex_indices" is a list of ints
    }
end_header                 { delimits the end of the header }
0 0 0                      { start of vertex list }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3                  { start of face list }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

Listing A.1: Example: PLY file describing a unit cube

Appendix B

Princeton Shape Benchmark classification

```
PSB 1
7 907

vehicle 0 230
1353
1361
(...)

animal 0 123
5
6
7
16
(...)

household 0 219
1598
1599
1602
(...)

building 0 53
382
383
384
(...)

furniture 0 104
944
945
948
(...)

plant 0 78
```

```
974
975
976
(...)
```

Listing B.1: Example: CLS file with six different classes. Object id list has been shortened for brevity.

Appendix C

Luxrender outputs

Output name	Bit depth
RGB	HDR
RGBA	HDR
RGB_TONEMAPPED	LDR
RGBA_TONEMAPPED	LDR
ALPHA	LDR
DEPTH	HDR
POSITION	HDR
GEOMETRY_NORMAL	HDR
SHADING NORMAL	HDR
MATERIAL_ID	LDR
DIRECT_DIFFUSE	HDR
DIRECT_GLOSSY	HDR
EMISSION	HDR
INDIRECT_DIFFUSE	HDR
INDIRECT_GLOSSY	HDR
INDIRECT_SPECULAR	HDR
MATERIAL_ID MASK	LDR
DIRECT_SHADOW MASK	LDR
INDIRECT_SHADOW MASK	LDR
RADIANCE_GROUP	LDR
UV	LDR
RAYCOUNT	HDR, LDR
BY_MATERIAL_ID	HDR, LDR
IRRADIANCE	HDR
OBJECT_ID	HDR
OBJECT_ID_MASK	HDR, LDR
BY_OBJECT_ID	HDR, LDR

Table C.1: Meta output supported by Luxrender