Physically-based rendering: A piece of cake

Jan Špaček

May 2016

1 Introduction

We present the design and implementation of `dort`[1], a physically-based renderer. `dort` is heavily influenced by `pbrt`, the renderer described in [4], but it has some important features on its own.

We will present the basic architecture of the system, its relation to `pbrt` and our own extensions. We will conclude with some remarks related to the "soft" aspects of software development.

2 Design

To render an image of the scene, the renderer must estimate the amount of light that arrives from the scene to the camera. This is accomplished by tracing rays from the camera, determining their intersections with the surfaces in the scene and computing the light reflected at the point of intersection in the direction of the ray. The formulas that describe the distribution of light in the scene involve integrals that cannot be computed in closed form, so we need to use Monte Carlo techniques to estimate these quantities numerically.

2.1 Geometry

The scene to be rendered is composed from primitives, which are represented in `dort` as objects derived from the class `Primitive`. Every primitive must implement a method that checks if a ray intersects the primitive and returns a structure describing the intersection (most notably the local geometry at the point of intersection, including the normal and derivatives of the local (u, v)-parameterization of the surface).

The intersection structure also contains a reference to a primitive which can be queried for the material properties of the intersection. There is a subclass of `Primitive`, named `GeometricPrimitive`, which adds the corresponding methods.

The simplest primitive is a `ShapePrimitive`, which contains a `Shape` and places the shape in the scene. The shape can be deformed and positioned using an arbitrary transformation matrix. The material properties of the primitive are determined by an instance of `Material`.

There are also primitives that cannot be intersected on their own (are not `GeometricPrimitive`s), but contain other primitives. This allows us to implement acceleration structures, which store the children primitives in a data structure that supports efficient spatial queries. At the moment, `dort` contains an implementation of primitives based on bounding-volume hierarchies (BVH).

The `Primitive` abstraction is one of the key design decisions inherited from `pbrt`. The unified treatment of acceleration structures and the scene geometry makes it possible to cleanly and efficiently separate geometry and shading.

[1] The code is available at `https://github.com/honzasp/dort` and is released into the public domain.

Figure 1: A scene rendered by `dort`, showcasing the support for Minecraft worlds.

2.2 Spaces and instancing

An important feature that must be supported by the geometric representation is instancing – the ability to include the same primitive in the scene multiple times at different locations. Scenes often contain duplicate objects (for example, in a model of a theatre, we can use a single model of a chair placed in the scene at hundreds of places), so substantial amounts of memory can be saved if a single copy of the object is referenced in the scene multiple times.

The primitive that is used as a placeholder for another primitive is named `FramePrimitive`. It defines a new space (coordinate system) for the referenced primitive by a transform matrix that maps the coordinates from the outer space to the inner space and back. A frame can contain other frames, so the spaces form a hierarchy. When a ray is traced, we compute the local geometry of the intersection (including the point of intersection) relative to the space where the intersected `GeometricPrimitive` is defined. However, we also need to compute and store the geometry relative to the world space (for example to compute the lighting, which is defined in the world space).

2.3 Shading

To compute the amount of light leaving a point on the surface of an object, we must know the reflective and transmissive properties of the surface. These are captured in a bidirectional scattering distribution function (BSDF), which maps an incident direction ω_i and exitant direction ω_o to the fraction of light arriving from ω_i that is reflected or transmitted in the direction of ω_o. The BSDF is stored as a sum of component functions, which are referred to as bidirectional reflection distribution functions (BRDFs) or bidirectional transmission distribution function (BTDFs), depending on whether they describe reflection or transmission.

`dort` implements the basic BRDFs and BTDFs such as perfectly diffuse (Lambertian) reflection, perfectly specular relection (mirror) and transmission (glass), Oren-Nayar microfacet BRDF and a generic microfacet BRDFs and BTDFs based on the framework described in [6].

When a primitive needs to supply a BSDF at a particular point, it usually delegates the request to an instance of `Material`, which creates a BSDF (represented with a `Bsdf` structure) based on the type of material. The materials implemented in `dort` include plastic (perfectly diffuse BRDF with a glossy microfacet BRDF), metal (a reflective microfacet BRDF) or glass (specular reflection and refraction, modulated with Fresnel coefficients). However, there is a big

(a) Plastic **(b)** Metal **(c)** Glass **(d)** Mirror

Figure 2: A statue of Happy Buddha rendered with different materials. The model is courtesy of Stanford Computer Graphics Laboratory.

room for experimentation with the various BRDFs described in the literature to implement interesting new materials or improve the existing ones.

2.4 Texturing

All materials require some parameters that modulate the appearance (for example color, roughness or the refraction coefficient). These parameters are supplied by textures, which are arbitrary spatially varying functions. The textures in `dort` are very generic, taking arbitrary input and producing arbitrary output.[2] Textures can be composed or combined in many other ways familiar to functional programmers.

The texturing system was designed to support powerful procedural textures (inspired by [2]). However, `dort` currently implements only the most basic procedural textures, such as value noise or a checkerboard pattern. The extension of the texturing system is an exciting prospect of further development.

2.5 Acceleration structures

There are basically four classes of data structures that are used to accelerate intersection queries: grids, octrees, kd-trees and BVHs. Grids divide the space into a regular grid of cells, octrees divide the space into an adaptively refined grid, kd-trees partition the space with axis-aligned planes and BVHs partition the primitives by their bounding volumes (usually axis-aligned boxes). We use BVHs as the main acceleration structure in `dort`, chiefly because their implementation is a bit simpler than kd-trees.

The BVH is a binary tree, where every node stores a bounding box of all the primitives contained in the subtree and leaves contain the primitives. The nodes can be stored efficiently in an array, requiring only 32 bytes per node (most of which is occupied by the bounding box). The leaves generally contain multiple primitives, the number of which can be controlled by the user. Smaller leaves generally mean that fewer triangles are tested for ray intersections, but the depth of the tree and the number of nodes is larger, consuming more memory. On the other hand, large leaves require testing more triangles, but the tree is shallower, so the number of nodes is smaller.

The main improvement of `dort` over `pbrt` is in the algorithm for building the tree. The approach given in `pbrt` first builds an ineffective representation of the tree using pointers, and then linearizes the tree into an array. The building is also not parallelized. When experimenting

[2]The texture class is nearly isomorphic to `std::function<A(B)>`, but our `Texture` can be a bit more efficient, because it never needs to be copied.

with large meshes, we often found that the running time of the BVH build exceeded the rendering itself, and the build required too much memory. For this reason, we implemented a parallelized building algorithm [3, 5] that builds the efficient representation directly.

2.6 Triangles and meshes

`dort` supports a wide range of geometric shapes, including spheres, disks, cylinders and cubes. However, models are in practice composed nearly exclusively from triangle meshes, and other shapes can be approximated by triangles quite well. Therefore, it is not unreasonable to design a renderer with support only for triangles, improving performance by using specialized algorithms and data structures.

We decided to keep the elegance of `pbrt` and retain first-class support for non-triangle shapes. We do not intend to use `dort` in practice, but we do need sample scenes to test the rendering algorithms, and we often find non-triangles very useful in defining these scenes.

To support meshes efficiently without sacrificing the overall design, `dort` extends on `pbrt` and includes several triangle primitives tuned for different needs. While it would certainly be possible to add each triangle as a `ShapePrimitive` referencing a triangle shape, this representation would be quite wasteful. A `ShapePrimitive` stores two 4-by-4 matrices, requiring 128 bytes of storage. The triangle shape would also need to store at least the three vertices (and maybe also (u, v)-mapping), giving more than $128 + 36 = 164$ bytes per a single triangle. As meshes can contain millions of triangles, we certainly need a better representation.

We can exploit the fact that multiple triangles share a single point in a triangle mesh, store all the points together in an array and use a single index instead of three floating-point coordinates. Moreover, we can also store the triplets of indices in an array, leaving us with only a pointer to the mesh structure and single integer index per triangle. The `Mesh` structure stores three floats per point and three integers per triangle. As the meshes contain approximately 0.5 points per triangle, we require approximately 18 bytes for each triangle in the mesh.

Having devised an efficient representation for meshes, we can turn our attention to how to represent the triangles with primitives. The first option is to use a `TriangleShape`, which references a triangle in the mesh by an index. We still need to use a `ShapePrimitive` to place the primitive in the scene, and intersecting the triangle amounts to two virtual calls (in the primitive and in the shape). Still, this `Shape` is useful at times, for example when defining area lights.

The second option is to use a hybrid of `ShapePrimitive` and `TriangleShape`, a `TriangleShapePrimitive`. This primitive does not store a transformation, refers directly to the mesh and saves a virtual call, but still needs to store a fat pointer[3] to the `Material`. Howerer, this representation makes it possible to specify different materials for different triangles, which is often desirable.

The third option, `MeshTrianglePrimitive`, stores the reference to `Material` together with the `Mesh` in a single structure, `PrimitiveMesh`, reducing the footprint of the primitive to a single pointer and an index[4]. This representation is already quite effective, but we still have to pay the overhead of virtual calls for every triangle tested for intersection.

The last option is a `MeshBvhPrimitive`. This primitive is a specialized aggregate that builds a bounding volume hierarchy from the triangles. The pointer to the mesh and the material is stored only once in the aggregate, and the triangles in the tree are represented as integer indices. The main benefit is that we can directly test the triangle intersections in the leaves, avoiding the virtual calls, and do not have to allocate a `Primitive` for every triangle. The drawback of this approach is that the triangles are not part of the aggregate that holds other primitives in the scene, which can hurt performance a bit, as the trees may not be in the optimal shape. We have not investigated this effect in detail, but we suspect that the reduce in virtual calls and memory consumption pays off overall.

[3] We use `std::shared_ptr`s to reference materials, but raw pointers for `Mesh`es, storing a `std::shared_ptr` to the `Mesh` in the `Scene` to keep the object alive. This strategy works well for meshes, because there are usually only a few of them, but storing pointers to all the materials in the `Scene` structure would be impractical.

[4] And a pointer to the virtual table.

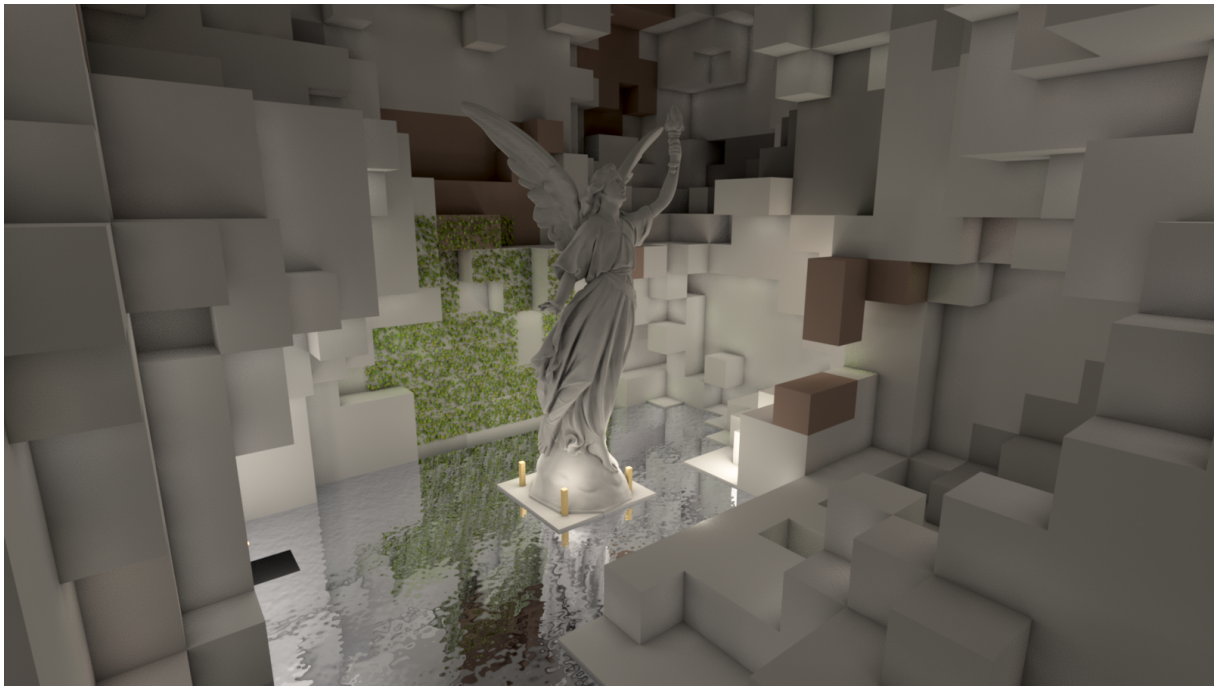

Figure 3: The statue of Lucy (model courtesy of Stanford Computer Graphics Laboratory) contains approximately 28 million triangles. This scene was rendered with the IGI algorithm using 64 samples per pixel.

2.7 Other components

Aside from the geometric and material description of the scene, a number of other objects is needed for rendering.

Lights are represented as objects derived from the class `Light`. `dort` currently supports point lights, diffuse area lights and uniform infinite lights. There are other useful types of lights, most notably image-based (or rather texture-based) environment lights, spotlights or texture projection lights, that may be implemented when (and if) they are needed.

Camera (`Camera`) is responsible for generating rays corresponding to pixels on the rendered image. `dort` currently implements orthogonal projection cameras and perspective cameras.

Images in `dort` are represented with class template `Image` parameterized on the type of the pixel, to support both 8-bit RGB images (low dynamic range) and 32-bit floating point RGB images (high dynamic range). Support for other formats, for example XYZ images, grayscale images or 16-bit half-floats, can be implemented easily. High dynamic range images are exported in the RGBE format (with extension `.hdr`), so that they can be tone-mapped in an external program. The support for tone mapping directly in `dort` is a possible future extension.

3 Lua scripting

One of the important aspects of the desing of a renderer is the way scenes are defined. `pbrt` uses a custom text file format to specify scenes, following the underlying representation quite closely. We decided to do better than `pbrt` and give the user the power of a real programming language for describing the scene. This SICP-inspired [1] decision allowed us to get the expressivity of a programming language and proceduralism essentialy for free.

Having suppressed the urge to design an implement our own language, we finally settled on Lua, mainly because of its convenient C bindings (Lua is primarily designed to be used as an extension language built into other applications) and ability to easily pass keyword arguments to functions via tables (hash maps). The only sad point about Lua is the convention of numbering arrays from 1 and inability to use conditionals as expressions (`if` is syntactically a statement and cannot return a value), otherwise we did not meet with a single issue.

We decided to give Lua all the power, so `dort` is essentially a large C++ library with Lua bindings. On startup, we initialize a Lua environment and execute the given program. The

Figure 4: A Minecraft jungle rendered by `dort` with the direct lighting algorithm using 16 samples per pixel.

program may then build a scene (or scenes), render it and write the result to an image file.

4 Minecraft scenes

Developing a renderer, it is necessary to have a source of interesting scenes to challenge the rendering algorithms. Rendering Stanford dragons, sculptures and spheres did not suffice for long, so we seeked other ways of obtaining non-trivial scenes. There are some publicly available models used throughout the graphics community, but unfortunately the file formats vary widely and are often closely tied to particular implementations. Probably the most common format is Wavefront OBJ, but reading these files is far from trivial and would require a large amount of tedious work.

In the end, we decided to implement reading worlds from Minecraft, a popular sandbox game. Worlds in Minecraft are procedurally generated, essentially unbounded, visually appealing and freely and easily editable. They exhibit a wide range of materials and models, making a perfect testbed for procedural texturing and modeling. The file format is also quite accessible and well documented.[5]

Reading Minecraft files and building the scene from the game world is a great match for Lua scripting. Lua is responsible for defining the voxels corresponding to Minecraft blocks, reading the blocks and building the scene. We implemented a specialized primitive, `VoxelGridPrimitive`, that provides very efficient rendering of voxels; however, nothing in the C++ code of `dort` relates directly to Minecraft.

The voxels in `VoxelGridPrimitive` are represented as 16-bit integers. Positive numbers denote solid cubes that completely fill the extent of the voxel, while negative numbers are voxels that contain a primitive. We devised a very efficient representation based on a kd-tree that always separates the space in the middle of the node, so we do not need to store the coordinate of the separating plane in the node. The voxels are stored in leaves of the tree, but we exploit the fact that large chunks of space are often composed of a single voxel (for example air or stone blocks usually form large structures) and allow leaves bigger than a single voxel. To render vast worlds with millions of voxels, the representation must be very memory efficient, so we pack the nodes of the tree into 4 bytes, requiring just 8 bytes per voxel in the worst case and much less in the usual case.

[5] Minecraft as a whole is probably one of the best documented piece of software we ever worked with.

5 Light transport algorithms

The light transport algorithm is at heart of every renderer. In `dort`, we implemented three of the algorithms described in [4]. The first is the direct-lighting renderer, which computes only the light that is directly reflected from the objects, without considering multiple reflection (indirect lighting).

The simplest rendering algorithm that does account for indirect lighting is path tracing, which traces random paths in the scene. However, in complex lighting conditions the random paths do not sample the light distribution well, leading to high variance in the estimate, which manifests as artifacts and noise in the resulting image.

We also implemented Instant global illumination (IGI), which uses a preprocessing step to trace random paths from light sources to create a set of virtual point lights, which are then used to estimate the indirect lighting.

Adding more advanced algorithms like photon mapping or Metropolis light transport is the most important way to extend `dort` in the future.

6 Remarks

Writing `dort`, we were pleasantly surprised how well did C++11 fare as the implementation language. We benefited heavily from having a carefully designed renderer as a starting point, so we avoided many iterations that would otherwise be necessary to come up with an architecture at least as half as good as we have now. The abstraction of primitives, shapes, materials and other components naturally leads to an object oriented representation. We also exploited the power of templates, most notably for the texturing system. We faced some amount of friction designing the Lua interface for the statically typed textures, because for operations like texture composition, we must explicitly handle every combination of texture types, quickly leading to combinatorial explosion.[6] Fortunately, a decent amount of template magic sufficed to generate the neccessary code with an acceptable level of boilerplate.

We developed `dort` iteratively, so that we could render images since the first commit, gradually extending and refactoring the system to enhance the functionality. The immediate feedback is invaluable not only because of the technical aspects, but also as an important motivational factor (we deliberately started working on a renderer with this aspect in mind).

7 Conclusion

Physically based rendering [4] is an outstanding book, serving both as a great introduction to the field of photorealistic rendering and as a reference implementation. Writing our own renderer has been a great and rewarding way to learn the basics of the subject, an experience that is unfortunately in stark contrast with most of the education we have encountered at schools.

There are innumerable ways how to improve `dort`, including new reflection models and materials, different types of lights or shapes, better light transport algorithms or novel acceleration structures. There are many ways how to utilize modern hardware, for example packet ray tracing with SIMD processing, computing on GPUs, and distributing the computations over multiple computers. There are also some optimizations, most notably in the ray-tracing of voxels, that could yield substantial improvements.

Bibliography

[1] Abelson, Harold, Gerald Jay Sussman, and Julie Sussman: *Structure and Interpretation of Computer Programs*. Second edition, 1996.

[2] Ebert, David S., F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley: *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, third edition, 2003.

[6] There are four valid texture input types and four output types, so for example texture composition breaks down to 64 cases.

[3] Lauterbach, Christian, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha: *Fast BVH construction on GPUs.* In *Computer Graphics Forum*, volume 28, pages 375–384, 2009.

[4] Pharr, Matt and Greg Humphreys: *Physically based rendering: from theory to implementation.* Morgan Kaufmann, second edition, 20010.

[5] Wald, Ingo: *On fast construction of SAH-based bounding volume hierarchies.* In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 33–40. IEEE, 2007.

[6] Walter, Bruce, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance: *Microfacet models for refraction through rough surfaces.* In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.