Implementing a Raytracer Using OpenCL

Introduction

Effectively using OpenCL to develop a raytracer involves overcoming a number of challenges that do not necessarily exist within the context of traditional raytracer implementations written in more conventional languages such as C++ or Java. These challenges are derived from restrictions that are necessary so that the programs written in the OpenCL language may fully harness the advantage of parallel systems and execute optimally, or, in some cases, at all.

In order to render photorealistic frames within reasonable timeframes, raytracers utilize random numbers, optimizing data structures and some kind of recursive execution that allows multiple rays to be bounced and/or scattered for each individual pixel. Within conventional raytracer implementations, random numbers are often accessible through the use of standard library functions, data structures are likely to make use of pointers (in one way or another) and the rays may be traced through the use of recursive functions.

Unfortunately, the OpenCL standard does not offer random number generators, support the passing of pointer-based data structures to parallel devices (referred to hereon in collectively as 'the device' or 'the devices') or allow for recursive calls to kernels within themselves. For this reason, alternative solutions must be written in order to facilitate the same functionality when executing programs in parallel and this document discusses potential suggestions for doing so.

Random Number Generation

Regardless of whether it is for it is for noise computation for texturing, Monte Carlo raytracing (path tracing) or artistic post-processing techniques, it is highly likely, if not certain, that an advanced raytracer will need to make use of randomly generated numbers at some point within the program.

Generating completely random numbers is not something supported by the OpenCL standard and, at current, there is no equivalent to C's 'rand' function in any implementation of the language. Despite this, it is entirely feasible to create functions that yield uniformly distributed random results when given a perceivably randomly generated seed. Dr. David B. Thomas (of Imperial College London) offers an implementation of such a random number generator in OpenCL that he, on his website, refers to as MWC64X (Thomas, David).

Despite the effectiveness of such generators, to produce different random numbers each time when used, the random states must first be initialized by some random function independent of the function used. This may be achieved through the allocation of an array on the host populated with random numbers (generated using whatever suitable random functions the host language may offer), that may then be passed to the device for use with the random number generator.

Thomas, David. (Date Unknown). *The MWC64X Random Number Generator.* [online] Accessed at <http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html> [Accessed 11th June 2013]

Building the Acceleration Structure

Although there are many valid methods for optimizing the evaluating of ray-to-triangle intersections within a three-dimensional scene (including binary space partitioning and grid hierarchies), in this document the implementation of a bounding volume hierarchy will be covered.

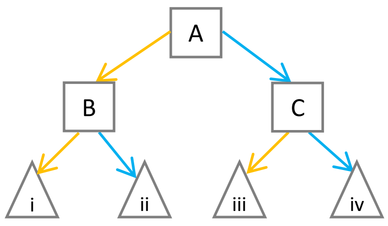
A bounding volume hierarchy (BVH) is essentially a binary tree in which the scene is divided into two three-dimensional boxes that, in turn, have increasingly smaller pairs of child boxes, until only single triangles fit within the boundaries of the child boxes. In this sense, all leaf nodes of a BVH are triangles and all other nodes are what may be referred to hereon in as ‘bounding boxes’.

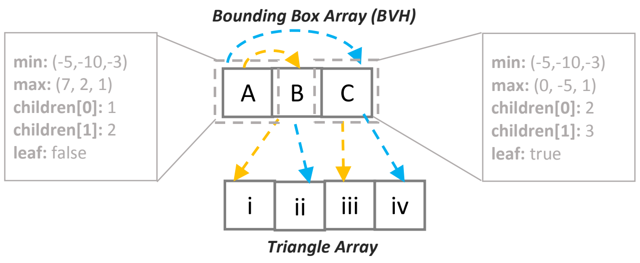
In object-oriented programming, a binary tree is something typically implemented through the use of dynamically allocated memory and pointers, but, given the number of leaf nodes may be pre-calculated simply be evaluating the number of triangles in the scene, it may be represented as an array that is no larger than the number of leaf nodes.

Although it is almost certainly feasible to create the BVH using OpenCL on the device, efficient allocation of workload and preventing data races could potentially be fairly complex difficulties to overcome and so, for the sake of simplicity, building an acceleration structure on the host is a reasonable approach.

Given that any part of the program written on the host does not share the restrictions of the OpenCL standard, it is possible to build a BVH recursively in a conventional manner, simply substituting pointer values for integer values representing indices of the array containing all bounding boxes. There are many ways in which the data representing the bounding boxes may be optimized so as to reduce memory footprint, but one of the most obvious refinements pertains to the nature of the overall BVH data structure itself.

As the triangles within a scene would also need to be stored in an array in order to be efficiently accessed on the device, the child members of the bounding boxes may be used to lookup in either the triangle or bounding box array without even having to cast the values used. Evidently, within the traversal of the BVH, the program would need to differentiate between leaf nodes and non-leaf nodes, but this may be easily achieved through the use of Boolean flags. The diagrams below convey this concept:





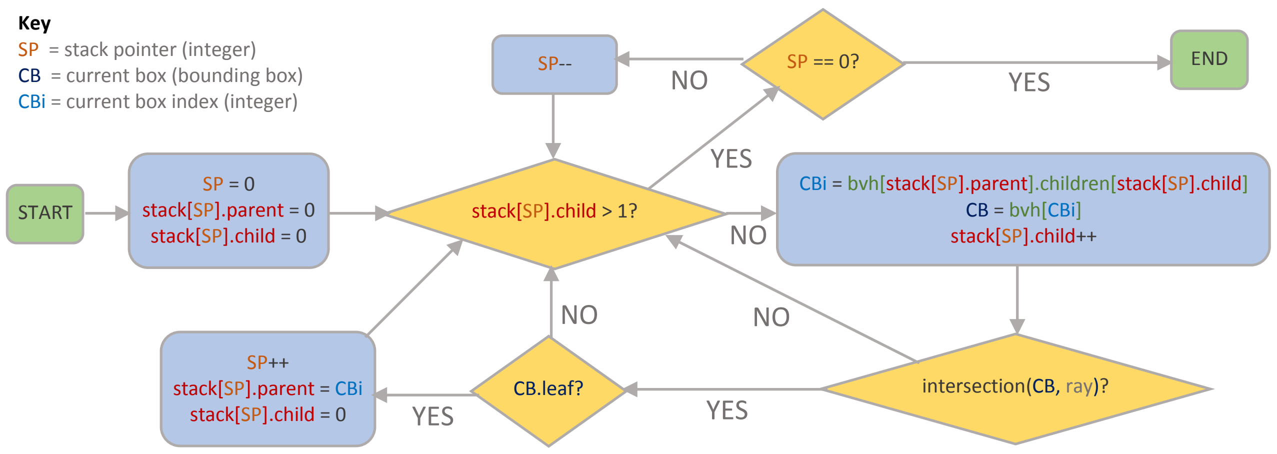
Traversing the Acceleration Structure

Although it is reasonable to build an acceleration structure on the host when implementing a raytracer using OpenCL, assuming the component of the program that performs the raytracing does so heterogeneously, it is essential to design the intersection routine so that the acceleration structure may be traversed the on the device, in parallel.

Again, using a BVH as an example of such a structure, where its building is complicated slightly by the need to use arrays in place of pointer-based systems, its traversal is made difficult by the lack of recursive calls within kernels. Within a BVH, it is necessary, in some way, to devise an implementation of a depth-first search algorithm for a binary tree, which is relatively straightforward using traditional recursive techniques, but the limitations of OpenCL make doing so more complex.

Although there are several possible stack-less methods for traversing a binary tree, one possible algorithm centers on the implementation of a rudimentary stack used to track the nodes the raytracer has evaluated and those which remain unevaluated.

The structure of the stack may be designed in many different ways, but one of the most simplistic, but still effective, approaches makes use of an array of elements that specify the index of the parent bounding box being evaluated and a second value representing whether it is the left or right child of the parent box being inspected at any given time. An integer may be used to represent the stack pointer so that the program may move up and down the stack as necessary. As OpenCL does not facilitate the dynamic allocation of memory on the device, in order to ensure an appropriate stack size for the BVH, it is necessary to write the kernel with a sufficiently large predefined array size hardcoded into the precompiled code, based upon the maximum expected depth of the BVH. The following flowchart demonstrates the potential execution patterns of a simple BVH stack:



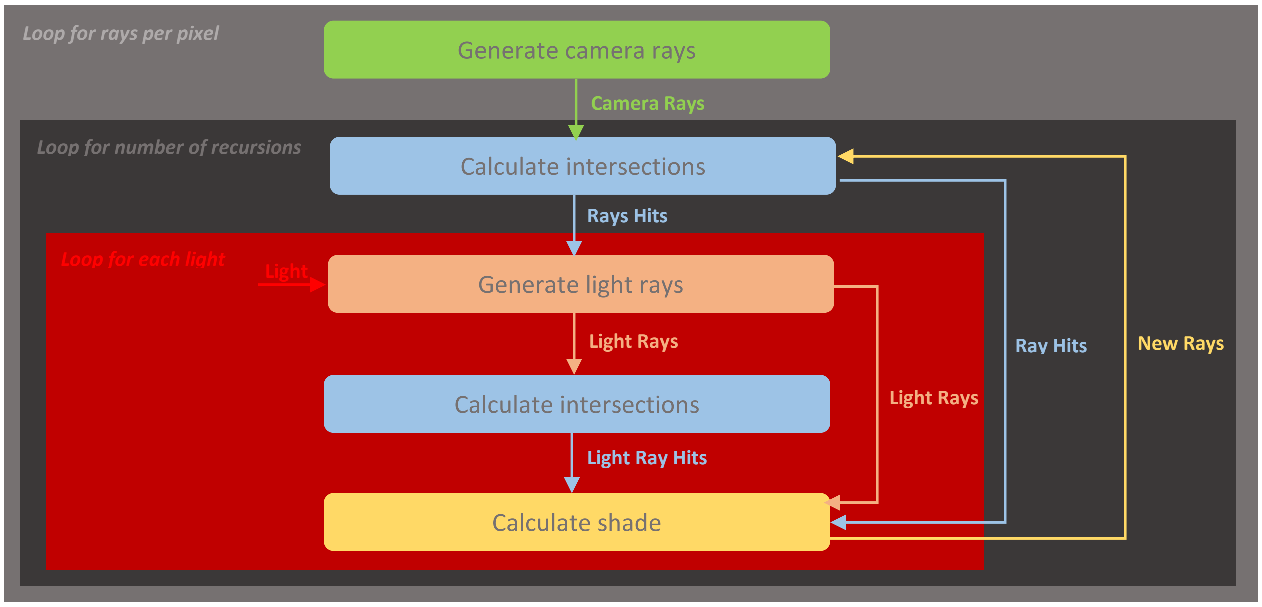
It is important to note that this design does not sort the child bounding boxes (based on which would be intersected first) before evaluating them, meaning that all potential triangle intersections must be considered before returning a result. Designing the stack system in such a way that would facilitate such sorting would be a major optimization, and is certainly something worthy of consideration.

Multiple Rays per Pixel

Raytracers that go beyond the Blinn-Phong shading model inevitably require more than one ray emission per pixel. Whether this is for reflection, refraction or path-tracing techniques, the program will need to offer some means by which multiple rays may contribute to the color of a single pixel.

In typical implementations, this may be achieved through the use of 'intersection' and 'shade' functions that may be called recursively whenever they require new rays to be emitted, but this is more complex in the context of OpenCL given that kernel recursion is not possible and many pixel values are calculated simultaneously in parallel, each of which may require a different number of rays depending on the surfaces with which they intersect.

For this reason, emitted ray intersections must be calculated iteratively, through the use of multiple calls from the host to kernels on the device that compute the intersection and shade results for each ray. Although the number of rays contributing to each pixel may vary greatly, the most straightforward approach involves maintaining and updating a fixed-size (the number of pixels) array of rays for each iteration, ensuring that only "valid" (which is to say, rays that have been emitted) are used to contribute to the respective pixel values. The following diagram is an indication of the program flow in one possible implementation of a simple raytracer:



Within this design, the intersections kernel is used twice – once for determining the first points in three-dimensional space that the emitted rays intersect and second for determining the first points in three-dimensional space that light rays intersect when pointing toward the respective hit points. These pairs of hits points may then be compared (for each pixel, in parallel) within the kernel calculating shade and, should they be the same, the emitted light ray is used to determine the shading of the surface intersected.