PARALLEL RENDERING GRAPHICS ALGORITHMS USING OPENCL

Gary Deng
B.S., California Polytechnic State University, San Luis Obispo, 2006

PROJECT

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

FALL
2011
PARALLEL RENDERING GRAPHICS ALGORITHMS USING OPENCL

A Project

by

Gary Deng

Approved by:

__________________________________, Committee Chair
John Clevenger, Ph.D.

__________________________________, Second Reader
V. Scott Gordon, Ph.D.

______________________________
Date

ii
Student: Gary Deng

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

__________________________, Graduate Coordinator       ________________
Nikrouz Faroughi, Ph.D.       Date

Department of Computer Science
Abstract

of

PARALLEL RENDERING GRAPHICS ALGORITHMS USING OPENCL

by

Gary Deng

The developments of computing hardware architectures are heading in a direction toward parallel computing. Whereas better and faster CPUs used to mean higher clock rates, better and faster CPUs now mean more cores per chip. Additionally, GPUs are emerging as powerful parallel processing devices when computing particular types of problems. Computers today have a tremendous amount of varied parallel processing power. Utilizing these different devices typically means wrestling with varied architecture, vendor, or platform specific programming models and code.

OpenCL is an open-standard designed to provide developers with a standard interface for programming varied (heterogeneous) parallel devices. This standard allows single source codes to define algorithms to solve vectorized problems on various parallel devices on the same machine. These programs are also portable.

This project explores OpenCL to implement a cross-platform, parallel solution to a vectorized problem. The domain of the problem is ray-tracing. Ray-tracing is a computer graphics
rendering algorithm that determines how to visualize a scene. A significant number of calculations are performed to colorize each pixel based on the data of the 3D objects in the scene. Though heavy, the calculations for each pixel can be made completely independently from the calculations of any of the other pixels.

The project has a GUI implemented in C++. The project has a ray-tracing engine implemented in C++. The ray-traced rendering routines come in four implementations: 1) written recursively in C++, executed on the CPU cores; 2) written iteratively in C++, executed on the CPU cores; 3) written iteratively in OpenCL C, executed in parallel on the CPU cores; 4) written iteratively in OpenCL C, executed in parallel on the GPU cores. The GUI reports the running time to perform the ray-tracing calculations to visualize the scene to the frame buffer for each ray-tracing implementation.
ACKNOWLEDGMENTS

I would like to express my gratitude to my project advisor Dr. John Clevenger who has taught me almost all I know about computer graphics.

I would like to give thanks to my second reader Dr. V. Scott Gordon for his assistance.

I would like to give my love and appreciation to my parents Yen-Hsi and Shu-Hsun who encouraged me to not give up when I just really, really wanted to.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>Page vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xiii</td>
</tr>
</tbody>
</table>

**Chapter**

1. **INTRODUCTION** ........................................................................................................ 1

2. **RELEVANT TOPICS AND TECHNOLOGIES** ................................................................  4
   2.1 Ray-Tracing ........................................................................................................ 4
   2.2 OpenCL ................................................................................................................ 8
   2.3 OpenCL and Vectorized Problems ....................................................................... 9
   2.4 OpenCL C Programming Language ....................................................................... 11
   2.5 JUCE Library ..................................................................................................... 12

3. **METHODOLOGY** ..................................................................................................... 13
   3.1 Application Overview ....................................................................................... 13
   3.2 Hardware and Operating System ....................................................................... 14
   3.3 The Engine ........................................................................................................ 15

4. **IMPLEMENTATION** ................................................................................................. 20
   4.1 Primitives, Materials, and Colors .................................................................. 20
   4.2 Ray-Tracing ....................................................................................................... 20
   4.3 GUI ....................................................................................................................... 26

5. **RESULTS AND CONCLUSIONS** ............................................................................. 35
LIST OF TABLES

Page

1. Table 4.1 The Four Ray-Tracing Implementations of the Project Application………21
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Figure 1.1 Amdahl’s Law Formula</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>Figure 2.1 Ray-Traced Scene from Whitted Article, Circa 1980</td>
<td>4</td>
</tr>
<tr>
<td>3.</td>
<td>Figure 2.2 Ray-Traced Scene by Gilles Tran Using POV-Ray Engine</td>
<td>5</td>
</tr>
<tr>
<td>4.</td>
<td>Figure 2.3 A Primary Ray Intersects with a Scene Object</td>
<td>6</td>
</tr>
<tr>
<td>5.</td>
<td>Figure 2.4 Traditional Ray-Tracing Routine</td>
<td>6</td>
</tr>
<tr>
<td>6.</td>
<td>Figure 2.5 Ray Tree Determines the Pixel’s Color</td>
<td>7</td>
</tr>
<tr>
<td>7.</td>
<td>Figure 2.6 The Vectorized Nature of Ray-Tracing for Each Pixel</td>
<td>8</td>
</tr>
<tr>
<td>8.</td>
<td>Figure 2.7 OpenCL Architecture</td>
<td>9</td>
</tr>
<tr>
<td>9.</td>
<td>Figure 2.8 OpenCL Platform Model</td>
<td>10</td>
</tr>
<tr>
<td>10.</td>
<td>Figure 2.9 Many Instances of One Kernel on One Device</td>
<td>11</td>
</tr>
<tr>
<td>11.</td>
<td>Figure 2.10 Parallel Execution on Similar Data</td>
<td>11</td>
</tr>
<tr>
<td>12.</td>
<td>Figure 3.1 Application’s Structure</td>
<td>13</td>
</tr>
<tr>
<td>13.</td>
<td>Figure 4.1 General Rendering Loop</td>
<td>22</td>
</tr>
<tr>
<td>14.</td>
<td>Figure 4.2 Recursive Ray-Trace Routine for One Pixel</td>
<td>23</td>
</tr>
<tr>
<td>15.</td>
<td>Figure 4.3 Iterative Ray-Trace Routine for One Pixel</td>
<td>25</td>
</tr>
<tr>
<td>16.</td>
<td>Figure 4.4 Initial State of GUI</td>
<td>27</td>
</tr>
<tr>
<td>17.</td>
<td>Figure 4.5 Viewing Window During Scene Rendering</td>
<td>28</td>
</tr>
<tr>
<td>18.</td>
<td>Figure 4.6 Viewing Window with Completely Rendered Scene</td>
<td>29</td>
</tr>
<tr>
<td>19.</td>
<td>Figure 4.7 Timing Display Before Scene Rendering</td>
<td>30</td>
</tr>
<tr>
<td>20.</td>
<td>Figure 4.8 Timing Display After Scene Rendering</td>
<td>30</td>
</tr>
<tr>
<td>21.</td>
<td>Figure 4.9 Implementation ComboBox with Selectable Values</td>
<td>31</td>
</tr>
<tr>
<td>22.</td>
<td>Figure 4.10 Render Button</td>
<td>32</td>
</tr>
</tbody>
</table>
43. Figure 5.16 Time vs. Run Among Implementations Without Restart:
   Maximum Ray-Trace Depth = 16............................................................... 52

44. Figure 5.17 Time vs. Run Among Implementations With Restart:
   Maximum Ray-Trace Depth = 0..................................................................... 53

45. Figure 5.18 Time vs. Run Among Implementations With Restart:
   Maximum Ray-Trace Depth = 16............................................................... 53

46. Figure 5.19 Time vs. Run between Serial CPU Implementations With Restart:
   Maximum Ray-Trace Depth = 0..................................................................... 54

47. Figure 5.20 Time vs. Run between Serial CPU Implementations With Restart:
   Maximum Ray-Trace Depth = 16............................................................... 55

48. Figure 5.21 Time vs. Run between Iterative Implementations With Restart:
   Maximum Ray-Trace Depth = 0..................................................................... 56

49. Figure 5.22 Time vs. Run between Iterative Implementations With Restart:
   Maximum Ray-Trace Depth = 16............................................................... 56

50. Figure 5.23 Time vs. Run between OpenCL Implementations With Restart:
    Maximum Ray-Trace Depth = 0..................................................................... 57

51. Figure 5.24 Time vs. Run between OpenCL Implementations With Restart:
    Maximum Ray-Trace Depth = 16............................................................... 58
LIST OF ABBREVIATIONS

1. OpenCL: Open Computing Language
2. OpenGL: Open Graphics Language
3. JUCE: Jules’ Utility Class Extensions
Chapter 1

INTRODUCTION

Parallel computing is everywhere. Traditionally considered “high-end computing,” the last 20 years have presented faster networks, distributed systems, and multi-processor computer architectures [1]. Today, many modern desktop or laptop machines house a multi-core Intel or AMD processor and possibly house discrete NVIDIA or ATI graphics processors with hundreds of processing cores. Even some gaming machines possess cell processors capable of parallel computing tasks. Hardwares capable of parallel computations are affordable and ubiquitous.

The proliferation of parallel computing hardware in pedestrian machines means that parallel capable resources are widely available; yet these resources are not so programmatically accessible. Writing parallel programs has always presented complexities; programmers must consider data dependencies, race conditions, communications, etc. As the available parallel capable computing resources go beyond just the CPU, even more programming complexities arise. To access all available resources, the same routine or algorithm may need to be coded multiple times and in multiple ways. The programmer must consider various type-, vendor-, or platform-specific programming models and/or APIs. As an example, coding a parallel program for a machine with an Intel multi-core CPU and one NVIDIA GPU would require a logically single parallel routine to be coded in two implementations: 1) C with OpenMP directives (for the CPU) [2] and 2) CUDA C (for the GPU) [3]. Furthermore, designing this program to be flexible enough to run on a system with an ATI GPU would require a third implementation of the routine using the Stream SDK [4] to accommodate the ATI GPU.

Writing a single source code base that accommodates the myriad of devices that may be present
in today’s machines may be demanding. Developer resources are wasted: excessive cognitive overhead is required in understanding various programming models, time is wasted in writing multiple but logically redundant code implementations. Furthermore, the code may have portability issues unless all possible parallel capable devices are programmatically accounted. A better way is needed. In 2008, Apple Computer proposed a draft specification for OpenCL (Open Computing Language) [5] to the Khronos Group [6]. The Khronos group explains that “OpenCL lets programmers write a single portable program that uses ALL resources in the heterogeneous platform” [7]. “OpenCL…is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors”. OpenCL presents programmers a standard API with which to program; participating vendors write their device drivers to conform to its specification. This abstraction enables programmers to focus less on device-specific programming details.

In this project, we investigate OpenCL by applying it to a parallel problem. Since one of the main motivations for parallel programming is to realize an increase in performance, it would be wise to choose a problem that can stand to benefit significantly in this way. Firstly, such a problem would have a large ratio of parallelizable:serialized code. Amdahl’s Law can be represented formulaically as [1]:
P is the fraction of code that can be parallelized. Maximum speedup of a program is directly proportional to the portion of the program that can be made parallel. Secondly, such a problem ought to be a vector problem. The OpenCL Specification indicates that its execution model is tailored toward handling vectorized problems; where single instructions of an instruction stream operate on vectors (A.K.A. arrays) of data as opposed to single instructions of an instruction stream operating on singular pieces of data. For our problem, we’ve chosen a classic algorithm from the domain of computer graphics rendering known as ray-tracing. This algorithm has a high proportion of parallelizable segments at run-time and is vectorized.

The software end-product of this project is a ray-tracing application which renders a defined-at-startup scene to an 800 x 600 resolution viewing window. Above this viewing window are GUI controls. The user may select one of four ray-tracing implementations, including OpenCL implementations, to choose how to render the scene and then begin to render it. When the implementation has finished its calculations and the scene has been rendered, the time elapsed to render the scene is superimposed over the visualization.
Chapter 2

RELEVANT TOPICS AND TECHNOLOGIES

2.1 Ray-Tracing

Whitted suggested the first general ray-tracing paradigm in 1980 [8]. Ray tracing is attractive because it can incorporate in a single framework: reflections of light, refractions of light, shadow computation, hidden surface removal, and global specular interactions [9]. Whereas many rendering algorithms sample only local surface data of the scene objects, i.e. only consider light source and surface orientations, this approach also considers global information to calculate intensities [8], i.e. reflections from other scene objects. The images produced are impressive but computationally expensive. Historically, considerable efforts have gone into investigating ways of overcoming these expenses [9].

Figure 2.1. Ray-Traced Scene from Whitted Article, Circa 1980 [8]
As with any computer graphics-rendering method, the goal is to generate a 2D image of a 3D scene (a collection of geometrically represented models) from the point of view of an observer (or camera) located at some position in the 3D scene.

In Jacco Bikker’s ray-tracing engine [11] [12], a rectangular grid model is placed at a fixed distance in front of the observer with the same dimensions as the final 2D image. To determine the color value for a single pixel, a primary ray is created that starts at the observer and continues through the section of the grid model that corresponds with the pixel of interest and into the scene. This primary ray may spawn child rays if it intersects with an object (model) in the scene.
Likewise, the child rays may spawn more rays as they further intersect with models. The final color of the pixel is determined by these series of intersections.

Starting with the primary ray, the following routine is executed:

<table>
<thead>
<tr>
<th>As a ray intersects with an object in the scene, two child rays may spawn:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) a ray that is reflected off of the surface of the object</td>
</tr>
<tr>
<td>2) a ray that is transmitted (and possibly also refracted) through the object</td>
</tr>
<tr>
<td>To determine local lighting calculations at an intersection, a shadow feeler is spawned toward each light source to test whether there is a direct path; i.e. no object exists between intersection and light source:</td>
</tr>
<tr>
<td>IF direct path exists: local light-surface contributions at intersection are made</td>
</tr>
<tr>
<td>ELSE no local light-surface contributions at intersection are made</td>
</tr>
<tr>
<td>Determination of global lighting calculations are a function of the total (global and local) lighting contributions of the intersection's child reflected and transmitted rays.</td>
</tr>
</tbody>
</table>

This process repeats with the child rays until a termination condition is reached:

1) the maximum depth has been reached, i.e. $n^{th}$-level child ray has been spawned.
2) a ray intersects with a light source
3) a ray goes off into space (does not intersect with any objects nor light sources)

4) the intersection point is determined to be in shadow (no direct paths to any light sources)

Because each ray intersection may spawn two more child rays, the process for a pixel may be conceptualized as creating a tree [8]. Each edge represents a reflected or transmitted ray and each node represents a ray-surface intersection and contributes to the final color of the pixel. The following figure illustrates an example of this concept of rays spawning more rays. Shadow feelers are omitted from the diagram. Though shadow feelers are similar in many ways, they differ from reflection and transmission rays as they do not spawn more rays upon intersecting with a surface. In other words, shadow feelers do not add more nodes to the “ray tree” in this model and are more appropriately conceptualized with the intersection calculations and samplings represented by the tree’s nodes.

![Ray Tree Diagram](image)

Figure 2.5. Ray Tree Determines the Pixel’s Color
For each pixel of the generated 2D image, this process (from primary ray, to generating the tree, to determining final pixel color) is executed. Because the routine for each pixel can be coded with the same sequence of instructions, this algorithm can be vectorized--the ray-tracing routine is parallelizable. Notice that at runtime, the proportion of parallelizable execution increases exponentially as maximum trace depth increases.

Figure 2.6. The Vectorized Nature of Ray-Tracing for Each Pixel

2.2 OpenCL

The *OpenCL Specification: Version 1.1* [7] states, “OpenCL is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.” It provides “an API for coordinating parallel computations across heterogeneous processors” and a “cross-platform programming language with a well-specified computation environment”.

OpenCL provides an abstraction for general purpose parallel computing across parallel-capable, heterogeneous devices. Instead of writing parallel programs with several device-specific APIs, programmers may instead define the problem once. The program’s parallel tasks are embodied in kernels using the OpenCL C language. Other portions of the application program use the OpenCL API to dictate how and on which capable devices of the host machine to execute the tasks defined by the kernels.

2.3 OpenCL and Vectorized Problems

The design of OpenCL supports operations on vectorized problems.
The OpenCL Platform Model’s highest level of abstraction is that of a *host* connected to one or more *compute devices*. Each compute device consists of one or more *compute units*. Within each compute unit are one or more *processing elements* (PE). Computations on a device occur within processing elements. Processing elements within a compute unit execute a single stream of instructions as SIMD units (execute in lockstep) or as SPMD units (each PE has its own program counter) [7].

When the host begins execution of a kernel, an *index space* is defined. For each point in this index space there exists an instance of the kernel. An instance of a kernel is known as a *work-item*. These work-items are grouped into *work-groups*, a coarse-grained decomposition of the index space. The work-items in a given work-group are executed concurrently on the processing elements of a compute unit. Each work-item executes the same code, but the execution pathway through the code and the operand data may vary [7].
Whether in lockstep or not, multiple instances of the same kernel are typically executed on vectors of similar data. As such, OpenCL’s design lends itself to computing vectorized problems.

2.4 OpenCL C Programming Language

OpenCL kernels are written using the OpenCL C Programming Language. It is based on the ISO/IEC 9899:1999 C language specification (a.k.a. C99 specification) but with extensions and
restrictions. Especially relevant is that recursion is not supported [7].

2.5 JUCE Library

According to its website, the JUCE library is an “all-encompassing C++ class library for developing cross-platform software [14].” The library can handle graphics and sound and provides tools for building customized GUIs. This includes displaying windows that can be drawn upon through OpenGL. The whole library can be incorporated into a project as two large source files. The JUCE library is released under the GNU Public License.
Chapter 3

METHODOLOGY

3.1 Application Overview

The project’s end product is a GUI-based software application that renders a ray-traced scene through one of four, user-selected, serial or parallel implementations of ray-tracing. Upon completion, the application also reports the time required to perform the ray-tracing calculations.

![Diagram of Application Structure](image)

Figure 3.1. Application’s Structure. Programming languages used are denoted in square brackets.
Libraries used are denoted in parentheses.

The Engine receives commands through the GUI, contains geometrical scene data, executes various ray-tracing routines (or arranges for their execution via OpenCL), and sends the rendered image data to the GUI for display to the computer screen.

The GUI starts the windowed application running. Users may instruct the engine of which ray-
tracing implementation should be used to render the scene. Users may instruct the engine to render the scene to an image with the currently selected implementation. At intervals, the GUI uses OpenGL calls to display the engine’s image data to a window. Upon completion, the GUI reports the engine’s execution time for rendering the scene.

The OpenCL Manager is the interface between the OpenCL kernel, OpenCL API, and the rest of the application. Upon initialization, the OpenCL Manager defines the context for the execution of the kernel. This includes:

1. Devices: OpenCL CPU and GPU devices to be used by the host.
2. Kernel: The ray-tracer function/routine to be run on the devices.
3. Program Objects: The program source and executable that implement the kernel.
4. Memory Objects: A set of memory objects containing data encapsulating the scene’s primitives that can be operated on by OpenCL devices and the host.

During runtime and as the application requests, the manager polls the context for and sets parameters such as OpenCL-determined optimal workgroup sizes for a device. Upon application shutdown, the manager frees OpenCL context resources.

3.2 Hardware and Operating System

The computing machine used to develop, to test, and to measure the performance of the software application for this project is an Apple MacBook Pro 17” laptop (no. A1261) with 4GB of RAM. The CPU is an Intel Core 2 Duo clocked at 2.5 GHz (no. T9300). The GPU is a GeForce 8600M GT with 512 MB of VRAM.

The operating system used to develop, to test, and to measure the performance of the software
application for this project is Mac OS X version 10.6.7. The IDE used for development is XCode version 3.2.6, with built-in support for Apple’s implementation of the OpenCL framework.

3.3 The Engine

The engine is based on one written in C++ by Jacco Bikker [11] [12] [15]. With his permission, it has been significantly modified for use in this project.

For this discussion, a *primitive* is the smallest, atomic, geometric object supported by the application.
**Original State**

Jacco Bikker’s engine [15] has these features:

- Graphics and drawing
  - Windows API calls
- Rendering routine
  - Ray-tracing
    - Ray casting
    - Reflection rays
    - Refraction rays
    - Supersampling of primary rays
  - Shadows
    - Shadow feelers
- Lighting calculations
  - Local
    - Based on material color data at ray-surface intersections
      - Diffuse shading
      - Specular shading
  - Global
    - Color of other surfaces reflecting from current ray-surface intersection
    - Refractive color through surface at ray-surface intersection
      - Based on Beer’s Law
  - Implementations
  - Serial
    - C++, recursive, CPU single-core

- Primitives
  - Planes
  - Spheres
  - Light sources - spherical
- Timing
  - Reports elapsed rendering time in minutes:seconds:milliseconds (MM:SS:mmss)
- Scene
  - Fixed Camera
  - Fixed Scene
    - 75 spheres
    - 3 planes
    - 2 light sources

**Modifications and Additions**

**Graphics and Drawing:**

The original engine was written for execution on the Microsoft Windows operating systems. Drawing the 2D rendered scene to the display window was originally accomplished by calls to the Windows API but have been replaced by OpenGL API calls. This change accommodates
execution on the Mac OS X operating system.

User Controls:
GUI controls were added using the JUCE library’s widgets. The original implementation had no user interaction other than opening and closing the application.

Additional Spheres and Reflective Planes:
Additional reflective spheres were added in order to make the rendering calculations more complex, i.e. more spawned reflection rays. In the same spirit, the back plane and bottom plane of the original scene were made to be reflective.

Ray-tracing Implementations:
Three ray-tracing implementations were added in accord with the four implementations described in section 4.2. Refraction rays are omitted from all four ray-tracing implementations (see discussion below).

Omissions
Refraction Rays and Supersampling:
Refraction rays are essential to traditional ray-tracing [8]; yet, were dropped from the ray-tracing implementations. The preorder traversals of binary ray trees required when both reflection and refraction rays are implemented are trivial with recursive ray-tracing algorithms, but are not so trivial with iterative versions. Because the latter three ray-tracing implementations in our project are required to be iterative, binary-tree traversals are avoided. Note that electing to keep refraction rays but dropping reflection rays would have achieved the same, simplifying effect.
Supersampling, or anti-aliasing, are described in Whitted’s ray-tracing paradigm [8]. To simplify the effort of developing three new, analogous implementations, these two details were not implemented.

Beer’s Law:

One feature in Jacco Bikker’s implementation of ray-tracing but not essential to traditional ray-tracing was lighting calculations using Beer’s Law. This extra detail was dropped.
**Current State**

The evolved ray-tracing engine has these features:

- Graphics and drawing
  - **OpenGL API calls**
- Rendering routine
  - Ray-tracing
    - Ray casting
      - Reflection rays
    - Shadows
      - Shadow feelers
    - Lighting calculations
      - Local
        - Based on material color data at ray-surface intersections
          - Diffuse shading
          - Specular shading
      - Global
        - Color of other surfaces reflecting from current ray-surface intersection
  - Implementations
    - Serial
      - C++, recursive, CPU single-core
      - C++, iterative, CPU single-core
    - **Parallel**
      - OpenCL C, iterative, CPU multi-core
      - OpenCL C, iterative, GPU
- Primitives
  - Planes
  - Spheres
  - Light sources - spherical
- Timing
  - Reports elapsed rendering time in minutes:seconds:milliseconds (MM:SS:msmsms)
- Scene
  - Fixed Camera
  - Fixed Scene
    - 83 spheres
    - 3 planes
    - 2 light sources
- **User Control**
  - GUI
Chapter 4

IMPLEMENTATION

4.1 Primitives, Materials, and Colors

The Plane and Sphere classes are the primitives for the engine and extend the Primitive class. All objects in the scene are a Plane or a Sphere. The Light class represents light sources in the scene and extends the Sphere class. Each Primitive contains a member Material class.

The Material class represents the properties of the material that a primitive is “made of.” A Material contains several float values such as diffuse, specular, reflection, and refraction coefficients in addition to a Color. The Color class contains three floats that define an RGB (red, green, blue) color.

4.2 Ray-Tracing

Four Ray-tracing Implementations

The application contains four ray-tracing implementations realized via three distinctly coded functions/kernels. The first implementation is defined by one C++ function; the second implementation is defined by another C++ function; the third and fourth implementations are defined by one OpenCL C kernel and external OpenCL API calls in the OpenCL Manager. The OpenCL API calls (in the host program) dictate whether to execute the kernel on the CPU or on the GPU.
<table>
<thead>
<tr>
<th>Number</th>
<th>Language</th>
<th>Compute Device</th>
<th>Compute Cores</th>
<th>Execution Style</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C++</td>
<td>CPU</td>
<td>Single core</td>
<td>Recursive</td>
<td>GCC CPU Recursive</td>
</tr>
<tr>
<td>2</td>
<td>C++</td>
<td>CPU</td>
<td>Single core</td>
<td>Iterative</td>
<td>GCC CPU Iterative</td>
</tr>
<tr>
<td>3</td>
<td>OpenCL C</td>
<td>CPU</td>
<td>Multi core</td>
<td>Iterative</td>
<td>OpenCL CPU</td>
</tr>
<tr>
<td>4</td>
<td>OpenCL C</td>
<td>GPU</td>
<td>Multi core</td>
<td>Iterative</td>
<td>OpenCL GPU</td>
</tr>
</tbody>
</table>

Table 4.1. The Four Ray-Tracing Implementations of the Project Application

Recursive, C++ Implementation:

Traditionally, ray-tracing implementations are recursive [8]. Traversing the binary ray tree for shading data is trivial since a recursive implementation’s execution flow mirrors a preorder traversal. Our first implementation reflects this tradition.

OpenCL Implementations:

Of particular interest are the OpenCL implementations of ray-tracing. The third and fourth implementations reflect this motivation of investigating OpenCL. As OpenCL C does not support recursion, these implementations are written iteratively. The work-group sizes used in these implementations are determined by querying OpenCL for its preferred work-group size via `clGetKernelWorkGroupInfo` method call [7].

Iterative, C++ Implementation:

Good comparisons require keeping as many variables constant as is possible among the subjects to be compared. In order to compare the OpenCL implementations to a single-core C++ implementation, a comparison ought to be made with a single-core C++ iterative implementation.
This is the motivation for the second implementation.

*Main Loop*

Regardless of the ray-tracing implementation, the main loop of the rendering routine can be described as in Figure 4.1.

For this discussion, a *pixel cohort* is a batch of pixels that are ray-traced before the routine moves on to update the display buffer. Please note that this value is independent of the OpenCL work-group size. Whereas a work-group is a coarse-grained decomposition of a problem as managed by OpenCL (refer to discussion in section 2.3), the pixel cohort size is the number of pixels to be processed by the ray-tracer before the GUI’s viewing window gets updated with the currently rendered scene image.

```
While there exists any unprocessed pixels in the output image
{
  /*Ray trace each pixel in the pixel cohort, assigning final RGB value to image data element
   *Send image data to video buffer to update display
  */
}
```

*Figure 4.1. General Rendering Loop*

How the portion of the routine at the asterisk is executed is defined by the selected ray-tracing implementation. Implementations #1 and #2 process each pixel in the cohort serially; one after the other. Implementations #3 and #4 process several pixels in the cohort simultaneously in workgroups via OpenCL.

To the user, the image is periodically refreshed after the engine finishes the rendering calculations for each pixel cohort.
Actual code for Figure 4.1 can be found in OpenGLWindow.cpp - DemoOpenGLCanvas::render() and ApplicationMain.cpp – Engine::Render(RenderModes mode) in Appendix A

*Recursive Implementation (Implementation #1)*

The traversal of the generated ray-tree is given “for free” with recursive implementations; i.e. the flow of control naturally follows a preorder traversal.

The process of ray-tracing a pixel begins with the creation of a primary ray for the pixel and a call to the ray-trace function; passing in the primary ray and a reference to an RGB color of (0,0,0) as parameters. The ray-trace method takes in a reference to a Ray object, a reference to a Color object, a ray depth, in addition to other parameters.

```
ray-trace ( Ray &ray, Color &rRGB, int depth ... )
{
   IF current ray-trace depth exceeds maximum depth: RETURN
   Find the nearest ray-surface intersection
   IF ray-surface intersection:
      IF Light source intersection: rRGB = (1,1,1)
      ELSE World object intersection:
         FOR EACH light source:
            Create shadow feeler from intersection to light source
            IF NOT feeler intersects with other world object:
               Update rRGB with local lighting contribution
            *IF surface is reflective, update rRGB by ray-tracing reflection ray
      }
}
```

Figure 4.2. Recursive Ray-Trace Routine for One Pixel

A reflection ray from the local surface is created and ray-traced via a recursive call to ray-trace() with an incremented depth, at the asterisk. The recursive ray-trace result provides lighting contributions from other surfaces reflecting onto the local surface.
Actual code can be found in ApplicationMain.cpp - Engine::Raytrace(Ray& a_Ray, Color &a_Acc, int a_Depth ...) in Appendix A

**Iterative Implementations (Implementations #2, #3, #4)**

For each pixel, there exist local variables tracking information about the Current Ray being traced and RGB (rRGB) values. Before iterating, the Current Ray’s variables are updated with the values of the primary ray and the RGB value is initialized to (0,0,0). Each iteration evaluates the Current Ray and updates the Current Ray’s data with a spawned ray (if applicable). Each iteration computes a local RGB value (lRGB) representing the Current Ray’s lighting contribution. This contribution is first weighted and then is used to update the pixel’s final RGB value (rRGB).

Weight-adjusted lRGB contribution:

In [16], Andrew D. Britton discusses a method to calculate the global lighting contribution from a series of successively spawned reflection rays in iterative implementations of ray-tracing that is equivalent to recursive implementations. The pixel’s final color can be described by the following equation:

\[
\text{color}_{\text{final}} = c_1(1 - x)x^0 + c_2(1 - x)x^1 + c_3(1 - x)x^2 + ... + c_n(1 - x)x^{n-1}
\]

: \text{color}_{\text{final}} \text{ is the pixel’s final color;}

\[c_n\] \text{ is the local lighting color of the surface intersecting with the } n^{th} \text{ spawned reflection ray;}

\[x^{n-1}\] \text{ is the reflectivity coefficient of the } n^{th} \text{ surface’s material raised to the } n-1 \text{ power;}

Notice that this equation assumes every surface’s reflectivity coefficient is the same; i.e. equal to \(x\). Britton also includes a \((1 - x)\) multiplier to \(c_n\). The author does not agree with this last detail
as it seems to imply that adding global reflection contributions detracts from a surface’s local lighting contribution.

Thus, the approach used in this project’s iterative implementations deviates slightly from Britton’s approach. The modification allows different surface materials to have different reflectivity coefficients. Furthermore, the (1-x) multiplier is omitted. The equation follows:

\[
\text{color}_{\text{final}} = c_1 \cdot 1 + c_2 \cdot x_1 + c_3 \cdot x_1x_2 + \ldots + c_n \cdot x_1x_2\ldots x_{n-1}
\]

: \text{color}_{\text{final}} \text{ is the pixel’s final color;}

\text{c}_n \text{ is the local lighting color of the surface intersecting with the } n^{\text{th}} \text{ spawned reflection ray;}

\text{x}_{n-1} \text{ is the reflectivity coefficient of the n-1}^{\text{th}} \text{ surface’s material;}

Thus, the weight-adjusted IRGB contribution for the \( n^{\text{th}} \) ray equals \( c_n \cdot x_1x_2\ldots x_{n-1}. \)

The iterative portion can be described as:

```plaintext
While ray-trace depth is less than or equal to maximum depth and not DONE
{
    Find the nearest ray-surface intersection
    IF ray-surface intersection:
        IF Light source intersection: IRGB = (1,1,1), DONE = TRUE
        ELSE World object intersection:
            FOR EACH light source:
                Create shadow feeler from intersection to light source
                IF NOT feeler intersects with other world object:
                    Update IRGB with local lighting contribution
                    IF surface is reflective, update “current ray” variables to reflection ray values
                    Adjust IRGB contribution (weighting) based on current ray-trace depth
            ELSE no ray-surface intersection: DONE = TRUE
    Add contribution of IRGB to rRGB
}
```

Figure 4.3. Iterative Ray-Trace Routine for One Pixel
Actual code can be found in ApplicationMain.cpp – Engine::Render(RenderModes mode) for Implementation #2 in Appendix A; raytracer.cl – raytrace(…) for Implementations #3 and #4 in Appendix B

4.3 GUI

The application’s GUI has a control menu and a viewing window. The upper, roughly 1/5 of the GUI consists of widgets to set the application’s behavior. The bottom, roughly 4/5 of the GUI is a window that presents the ray-traced rendered results of the scene.
Figure 4.4. Initial State of GUI

*Viewing Window*

Blank: Upon launch, the application presents a black, blank viewing window (except for “TIMINGS:" text).
Rendering: While the application is performing calculations to render the scene, the viewing
window will be periodically updated with the portion of the image, or pixels, that have completed
their ray-tracing processing.
Figure 4.6. Viewing Window with Completely Rendered Scene

Rendering Complete: When all pixels of the image have been rendered, the viewing window stops updating and displays the final, ray-traced, rendered image.

Timing Display: Upon launch, the timing display presents no value. After the scene has been completely rendered by any implementation, a value is presented. This value is in MM:SS:msms (minutes, seconds, milliseconds) format. The value represents the time elapsed between the
moment any ray-traced rendering processing began and the moment all ray-traced rendering processing had finished for the scene.

Figure 4.7. Timing Display Before Scene Rendering

Figure 4.8. Timing Display After Scene Rendering
**Controls**

![Implementation Combo Box with Selectable Values](image)

**Figure 4.9.** Implementation Combo Box with Selectable Values

Implementation Combo Box: Allows the user to select one of four ray-traced rendering implementations as described in Section 4.2. Selection does not begin rendering of the scene.
Render Button: Selecting the Render Button begins rendering the scene. The executed rendering implementation corresponds to that which is selected in the Implementation Combo Box.

Max Trace Depth Slider: The value on this slider sets the maximum trace depth for all rendering implementations. The maximum trace depth dictates how many levels of successive descendant reflection rays will be spawned from primary rays. The maximum trace depth can be set within an inclusive range of 0 through 16.

OpenCL Recommended Workgroup Size Label: The value at the rightmost portion represents the recommended workgroup size prescribed by OpenCL for the currently selected implementation.
using the \texttt{clGetKernelWorkGroupInfo} method call \cite{7}.

When the selected implementation does not involve OpenCL, the displayed value is “--”.

Figure 4.13. Pixel Cohort Size Combo Box with Selectable Values

Pixel Cohort Size Combo Box: This combo box sets the size of the pixel cohort that the application will use for rendering the scene. Several pre-determined values are presented.
Ray-Surface Intersections Label: Before the scene becomes rendered, the value for this label reads “--”. As the scene becomes rendered, the application keeps track of how many rays intersected with some surface and displays this value here.

Ray Misses Label: Before the scene becomes rendered, the value for this label reads “--”. As the scene becomes rendered, the application keeps track of how many rays did not intersect with some surface and displays this value here.
RESULTS AND CONCLUSIONS

5.1 Rendered Scene Images

*Rendered Images Among Maximum Ray-trace Depths*

The following are rendered images of the scene at progressively higher Max Trace Depth settings. Since the purpose is to show how the scene visually appears with increasing levels of Max Trace Depth, all other parameters remain constant (workgroup size is 1, as recommended by a query to OpenCL for the CPU device, and the pixel cohort size is 262144). The images are all rendered using the OpenCL CPU ray-tracing implementation. As demonstrated later in this chapter, all four ray tracing implementations render visually indistinguishable scenes when using the same parameters.
Figure 5.1. Rendered Scene: Maximum Ray-Trace Depth = 1
Figure 5.2. Rendered Scene: Maximum Ray-Trace Depth = 2
Figure 5.3. Rendered Scene: Maximum Ray-Trace Depth = 3
Figure 5.4. Rendered Scene: Maximum Ray-Trace Depth = 4
Figure 5.5. Rendered Scene: Maximum Ray-Trace Depth = 5
Figure 5.6. Rendered Scene: Maximum Ray-Trace Depth = 16

At a maximum ray-trace depth of 0, the rendered image is blank. At a maximum depth of 1, the rendered image has no reflections. At a maximum depth of 2, the rendered image shows reflective surfaces that show other objects in the scene (e.g. reflective spheres that present other spheres as non-reflective). At a maximum depth of 3, reflective surfaces show other reflective objects that further reflect other objects (e.g. reflective spheres that present reflective spheres). At a maximum depth of 4, some reflective surfaces become more complexly reflected (e.g. the
light reflection pattern between the purple back-plane and the large left-sphere). Images rendered with maximum ray-trace depths at 4 and above (through 16) do not visually appear to have any noticeable differences from one another.

The rendered images resulting from progressively higher maximum ray-trace depths from 0 through 4 is what one would expect. As maximum depth increases, the series of reflection rays permitted to globally sample colors increases. In other words, more reflections are observed on reflective surfaces and may become increasingly complex. However, depths of 4 and more do not present any visually observable increase in detail.

*Rendered Scenes Generated by the Four Implementations*

Ray-traced images at any particular level produce visually identical images among all four types of implementations at any specific maximum ray-trace depth.
Figure 5.7. Rendered Scenes Generated by the Four Implementations: Maximum Ray-Trace Depth = 1
Figure 5.8. Rendered Scenes Generated by the Four Implementations: Maximum Ray-Trace Depth = 2
Figure 5.9. Rendered Scenes Generated by the Four Implementations: Maximum Ray-Trace Depth = 3
Figure 5.10. Rendered Scenes Generated by the Four Implementations: Maximum Ray-Trace Depth = 4
5.2 Implementation Ray Statistics

The application keeps track of two metrics regarding rays: Ray-surface Intersections (the number of primary or reflection rays that intersect with some surface) and Ray Misses (the number of primary or reflection rays that do not intersect with some surface). Though the rendered images among all four implementations may appear to be visually indistinguishable, the ray statistics are not unanimously identical. The GCC CPU Recursive, GCC CPU Iterative, and OpenCL CPU
implementations produce scenes with identical statistics yet the OpenCL GPU implementation renders scenes with statistics that differ from the other three implementations.

*Ray-Surface Intersections Among the Four Implementations*

The number of ray-surface intersections for the OpenCL GPU Implementation are slightly greater than the number of ray-surface intersections for any of the other three implementations. The number of ray-surface intersections for the other three implementations (CPU Recursive, CPU Iterative, and OpenCL CPU) are the same among one another.

The number of ray-surface intersections at max trace depth of 16 is 1197468 for the OpenCL GPU Implementation. The number of ray-surface intersections at max trace depth of 16 is 1193437 for the other three implementations; a rather slight difference.

![Ray-Surface Intersections vs. Max Trace Depth](image)

Fig 5.12. Ray-Surface Intersections vs. Max Trace Depth
Ray-Surface Intersections vs. Max Trace Depth (Selected Data)

Ray Misses Among the Four Implementations

The number of ray misses for the OpenCL GPU Implementation is somewhat less than the number of ray misses for the other three implementations. The number of ray misses at max trace depth of 16 for the OpenCL GPU Implementation is 1838. The number of ray misses at max trace depth of 16 for the other three implementations is 2344.
It appears evident that the OpenCL GPU Implementation’s deviation in results is due to something specific to the GPU. The OpenCL CPU Implementation’s ray statistics match those of both the GCC CPU Recursive and Iterative Implementations’. This shows that the OpenCL Implementations’ kernel algorithm can produce the same results as the GCC CPU Implementations algorithms. The OpenCL GPU Implementation’s ray statistics deviate from those of the OpenCL CPU Implementation. As the OpenCL CPU and GPU Implementations share the same kernel code (program code), the deviations in results obtained suggest some factor specific to the GPU; whether possibly an issue in Nvidia/Apple’s implementation of the OpenCL framework for the GPU, or floating point precision limitations on the GPU, or some other factor.

5.3 Implementation Run Times

This section presents and examines data relating to run time measurements. A “run” begins when the selected ray-tracing implementation begins rendering the scene and ends when it has completed the rendering of the entire scene. The section starts off with an investigation of why
the application is restarted between runs. Run-time comparisons are then made between GCC
CPU Recursive and GCC CPU Iterative Implementations, followed by comparisons among the
three iterative implementations, and finally comparisons between OpenCL CPU and OpenCL
GPU implementations.

**Successive Runs without and with Application Restart**

Figures 5.15 and 5.16 present the run times without restarting the application between successive
runs. Notice that the OpenCL GPU implementation’s run times increase with successive runs
without restart.

Figure 5.15. Time vs. Run Among Implementations Without Restart: Maximum Ray-Trace Depth
= 0
Figures 5.17 and 5.18 present the run times with restarting the application between successive runs. Notice that restarting the application between runs eliminates the increase in run times observed when the application is not restarted between runs.
Figure 5.17. Time vs. Run Among Implementations With Restart: Maximum Ray-Trace Depth = 0

Figure 5.18. Time vs. Run Among Implementations With Restart: Maximum Ray-Trace Depth = 16

It is not clear why the OpenCL GPU Implementation’s run times increase with successive runs. XCode’s performance tool, Leaks, suggests that there are no memory leaks in the application. It
is within the realm of possibility that the problem lies in the implementation of the OpenCL framework for the GPU as implemented by Nvidia/Apple, since the OpenCL CPU implementation does not suffer the same issue; yet uses the exact same kernel code.

In order to avoid the phenomenon of increasing run times with successive runs, the comparisons that follow are measured with restarting the application between runs.

_Run Times Between Serial CPU Implementations_

The following figures show a comparison between the run times of the GCC CPU Iterative Implementation and the GCC CPU Recursive Implementation. Data are measured at a max trace depth of 0 (a state with the theoretically least amount of ray-traced processing) and at a max trace depth of 16 (a state with the theoretically most amount of ray-traced processing supported by the application).

![Figure 5.19. Time vs. Run between Serial CPU Implementations With Restart: Maximum Ray-Trace Depth = 0](image)
Comparing the two serial CPU implementations is a bit of an “apples and oranges” comparison. There are at least a few factors that differ between the two. As discussed in Chapter 2, the algorithms are fundamentally different. Secondly, recursive implementations are generally more memory intensive while iterative implementations are generally more processor intensive. Lastly, the recursive implementation uses OOP classes and constructs while the iterative implementation does not; instead using arrays to store and access data.

That being said, the serial CPU iterative implementation has better performance than the serial CPU recursive implementation.

**Run Times Among Iterative Implementations**

The following figures show a comparison between the iterative implementations: GCC CPU Iterative, OpenCL CPU, and OpenCL GPU. Data are measured at a max trace depth of 0 (a state
with the theoretically least amount of ray-traced processing) and at a max trace depth of 16 (a state with the theoretically most amount of ray-traced processing supported by the application).

Figure 5.21. Time vs. Run between Iterative Implementations With Restart: Maximum Ray-Trace Depth = 0

Figure 5.22. Time vs. Run between Iterative Implementations With Restart: Maximum Ray-Trace Depth = 16
Both OpenCL ray-tracing implementations present significantly lower run times than the serial CPU iterative ray-tracing implementation. The average run time for the serial CPU iterative implementation is about six to seven times greater than either of the OpenCL implementations.

It is interesting that the OpenCL CPU iterative implementation sees such a great performance gain when compared to the GCC CPU Iterative Implementation. Perhaps Apple’s OpenCL framework implementation is quite efficient at managing CPU/memory resources.

**Run Times between OpenCL Implementations**

The following figures show a comparison between the OpenCL implementations: OpenCL CPU and OpenCL GPU. Data are measured at a max trace depth of 0 (a state with the theoretically least amount of ray-traced processing) and at a max trace depth of 16 (a state with the theoretically most amount of ray-traced processing supported by the application).

![Time vs. Run between OpenCL Implementations at Max Depth = 0 (with restart)](image)

**Figure 5.23.** Time vs. Run between OpenCL Implementations With Restart: Maximum Ray-Trace Depth = 0
Figure 5.23 shows that the OpenCL CPU implementation has higher run times than the OpenCL GPU implementation at maximum ray-trace depth of 0. Because the kernel code for both implementations is the exact same, these data suggest that there exists an overhead in running the OpenCL CPU implementation that either does not exist or is more minimal when running the OpenCL GPU implementation.

Figure 5.24 shows that the OpenCL CPU implementation has higher run times than the OpenCL GPU implementation at maximum ray-trace depth of 16. Qualitatively, these results are in line with the expected results. Ray-tracing is a rather parallelizable task and dividing the task among more processing elements would logically lower the time needed to complete the task.

Quantitatively, the interpretation of these results is not so clear. The GPU has many times more processing elements than the CPU, yet the average run time of the OpenCL CPU implementation is 126% that of the average run time of the OpenCL GPU implementation. There may be many
hardware considerations that help the CPU to narrow the performance gap such as relatively long bus transfer times between the host program’s CPU and GPU processors and higher cache coherency in the CPU’s Level 1 cache.
Chapter 6

FUTURE WORK

6.1 Port to Other Platforms

It would be interesting to port the application over to other hardware and implementations of the OpenCL framework and see how the various ray-tracing implementations perform. When ported over to other implementations of the OpenCL framework, one could investigate whether or not successive runs of the OpenCL GPU Implementation continues to exhibit the gradual loss of performance as well as investigate whether the ray statistics for the OpenCL GPU Implementation continue to deviate from the other three. One could also investigate whether or not other hardware would yield similar results as we have observed.

6.2 OpenMP Versus OpenCL CPU Implementations

As previously mentioned, the significant improvement observed in the OpenCL CPU ray-tracing implementation over the serial CPU Iterative implementation was surprising. It might be interesting to develop other multi-core CPU iterative ray-tracing implementations like the ones in this project but using other APIs, such as OpenMP. Would one observe similar speedups or is there something about OpenCL’s management of multi-core CPUs that is very optimized and efficient?

6.3 Movable Camera and Animation

If given fast enough hardware and continuous re-rendering of the scene, could the OpenCL ray-tracing implementations in this project maintain multiple frames per second rates? In addition, could one implement a movable camera to give the appearance of moving around in a 3D, ray-
6.4 Refraction Rays

This project deviated from traditional ray-tracing by not including refraction rays, but only reflection rays. One could implement both types of rays and observe performance trends.
APPENDICES
APPENDIX A

Source Code in C++

```cpp
// ************************************************************************
// app_common.h
// 2011 - modified - Gary Deng - dengsta@gmail.com
// 2004 - original - Jacco Bikker - jacco@bik5.com - www.bik5.com - <>
// Common application definitions
// ************************************************************************

#ifndef I_APPLICATION_COMMON_H
#define I_APPLICATION_COMMON_H

#define DEBUG false

// Intersection descriptions/types (for emulating OpenCL iterative ray-tracing
implementation)
#define NO_HIT    0
#define PLANE_HIT 1
#define SPHERE_HIT_OUTSIDE 2
#define SPHERE_HIT_INSIDE 3

#include "math.h"
#include "stdlib.h"
#include <OpenCL/OpenCL.h>
#include <vector>
#include <iostream>

typedef unsigned int Pixel;

inline float Rand( float a_Range ) { return ((float)rand() / RAND_MAX) * a_Range; }

namespace Application {
    #define DOT(A,B)  (A.x*B.x+A.y*B.y+A.z*B.z)
    #define NORMALIZE(A){float l=1/sqrtf(A.x*A.x+A.y*A.y+A.z*A.z);A.x*=l;A.y*=l;A.z*=l;}
    #define LENGTH(A)  (sqrtf(A.x*A.x+A.y*A.y+A.z*A.z))
    #define SQRLENGTH(A)(A.x*A.x+A.y*A.y+A.z*A.z)
    #define SQRDISTANCE(A,B) ((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y)+(A.z-B.z)*(A.z-B.z))
    #define EPSILON   0.001f
    #define PI    3.141592653589793238462f

    class vector3
    {
    public:
        vector3() : x( 0.0f ), y( 0.0f ), z( 0.0f ) {};
        vector3( float a_X, float a_Y, float a_Z ) : x( a_X ), y( a_Y ), z( a_Z ) {};
        void Set( float a_X, float a_Y, float a_Z ) { x = a_X; y = a_Y; z = a_Z; }
        void Normalize() { float l = 1.0f / Length(); x *= l; y *= l; z *= l; }
        float Length() { return (float)sqrt( x * x + y * y + z * z ); }
        float SqrLength() { return x * x + y * y + z * z; }
        float Dot( vector3 a_V ) { return x * a_V.x + y * a_V.y + z * a_V.z; }
        vector3 Cross( vector3 b ) { return vector3( y * b.z - z * b.y, z * b.x - x * b.z, x * b.y - y * b.x ); }
        void operator += ( const vector3& a_V ) { x += a_V.x; y += a_V.y; z += a_V.z; }
        void operator += ( const vector3* a_V ) { x += a_V->x; y += a_V->y; z +=
```
void operator -= (const vector3& a_V) { x -= a_V.x; y -= a_V.y; z -= a_V.z;
void operator -= (const vector3* a_V) { x -= a_V->x; y -= a_V->y; z -= a_V->z;
void operator *= (const float f) { x *= f; y *= f; z *= f;
void operator *(const vector3& a_V) { x *= a_V.x; y *= a_V.y; z *= a_V.z;
void operator *(const vector3* a_V) { x *= a_V->x; y *= a_V->y; z *= a_V->z;
friend vector3 operator+(const vector3& v1, const vector3& v2) {
friend vector3 operator-(const vector3& v1, const vector3& v2) {
friend vector3 operator+(const vector3& v1, const vector3* v2) {
friend vector3 operator-(const vector3& v1, const vector3* v2) {
friend vector3 operator*(const vector3& v1, const vector3* v2) {
friend vector3 operator*(const float f, const vector3& v) { return vector3(v.x * f, v.y * f, v.z * f);

union {
    struct { float x, y, z; };
    struct { float r, g, b; };
    struct { float cell[3]; };
};

class plane {
    public:
        plane(vector3 a_Normal = vector3(0, 0, 0), float a_D = 0) : N(*((vector3*)cell)), D(cell[3]) {
            N = a_Normal;
            D = a_D;
        };
    float cell[4];
    vector3 &N;
    float &D;
};

typedef vector3 Color;

enum RenderModes {
    GCC_CPU_Recursive = 0,
    GCC_CPU_Iterative = 1,
    OpenCL_CPU = 2,
    OpenCL_GPU = 3
};

struct PixelDataStructOpenCL {
    size_t size;
    cl_mem color;
    PixelDataStructOpenCL(size_t a_Size) { size = a_Size; }
};
struct PlaneDataStructCpp
{
    size_t size;
    // Geometric data
    float* normalX;
    float* normalY;
    float* normalZ;
    float* d;  // Need this?
    // Material data
    float* diffuse;
    float* specular;
    float* reflection;
    float* refrIndex;
    // Color data
    float* red;
    float* green;
    float* blue;
    // Light data
    int* isLight;

    PlaneDataStructCpp(size_t a_Size = 0) {
        size = a_Size;
        normalX = new float[size];
        normalY = new float[size];
        normalZ = new float[size];
        d = new float[size];
        diffuse = new float[size];
        specular = new float[size];
        reflection = new float[size];
        refrIndex = new float[size];
        red = new float[size];
        green = new float[size];
        blue = new float[size];
        isLight = new int[size];
    }

    -PlaneDataStructCpp() {
        delete[] normalX; normalX = NULL;
        delete[] normalY; normalY = NULL;
        delete[] normalZ; normalZ = NULL;
        delete[] d; d = NULL;
        delete[] diffuse; diffuse = NULL;
        delete[] specular; specular = NULL;
        delete[] reflection; reflection = NULL;
        delete[] refrIndex; refrIndex = NULL;
        delete[] red; red = NULL;
        delete[] green; green = NULL;
        delete[] blue; blue = NULL;
        delete[] isLight; isLight = NULL;
    }
};

struct PlaneDataStructOpenCL {
    size_t size;
    // Geometric data
    cl_mem normalX;
    cl_mem normalY;
    cl_mem normalZ;
    cl_mem d;  // Need this?
    // Material data
    cl_mem diffuse;
    cl_mem specular;
    cl_mem reflection;
cl_mem refraction;
cl_mem refrIndex;
// Color data
cl_mem red;
cl_mem green;
cl_mem blue;
// Light?
cl_mem isLight;

PlaneDataStructOpenCL(size_t a_Size) {
    size = a_Size;
} -PlaneDataStructOpenCL() {
}

struct SphereDataStructCpp {
    size_t size;
    // Geometric data
    float* centerX;
    float* centerY;
    float* centerZ;
    float* recRadius;
    float* sqRadius;
    // Material data
    float* diffuse;
    float* specular;
    float* reflection;
    float* refractive;
    float* refrindex;
    // Color data
    float* red;
    float* green;
    float* blue;
    // Light?
    int* isLight;

    SphereDataStructCpp(size_t a_Size = 0) {
        size = a_Size;
        centerX = new float[size];
        centerY = new float[size];
        centerZ = new float[size];
        recRadius = new float[size];
        sqRadius = new float[size];
        diffuse = new float[size];
        specular = new float[size];
        reflection = new float[size];
        refractive = new float[size];
        refrindex = new float[size];
        red = new float[size];
        green = new float[size];
        blue = new float[size];
        isLight = new int[size];
    }

    -SphereDataStructCpp() {
        delete[] centerX;  centerX = NULL;
        delete[] centerY;  centerY = NULL;
        delete[] centerZ;  centerZ = NULL;
        delete[] recRadius;  recRadius = NULL;
        delete[] sqRadius;  sqRadius = NULL;
        delete[] diffuse;  diffuse = NULL;
        delete[] specular;  specular = NULL;
        delete[] reflection;  reflection = NULL;
        delete[] refractive;  refractive = NULL;
        delete[] refrindex;  refrindex = NULL;
        delete[] red;  red = NULL;
        delete[] green;  green = NULL;
    }
}
struct SphereDataStructOpenCL {
 size_t size;
 // Geometric data
 cl_mem centerX;
 cl_mem centerY;
 cl_mem centerZ;
 cl_mem recRadius;
 cl_mem sqRadius;
 // Material data
 cl_mem diffuse;
 cl_mem specular;
 cl_mem reflection;
 cl_mem refraction;
 cl_mem refrIndex;
 // Color data
 cl_mem red;
 cl_mem green;
 cl_mem blue;
 // Light?
 cl_mem isLight;
}

SphereDataStructOpenCL(size_t a_Size = 0) {
 size = a_Size;
}
~SphereDataStructOpenCL() {
}

struct StatisticsDataStructCpp {
 size_t size;
 // Statistics data
 int* rayIntersectionsCount;
 int* rayMissesCount;

StatisticsDataStructCpp(size_t a_Size = 0) {
 size = a_Size;
 rayIntersectionsCount = new int[size];
 rayMissesCount = new int[size];
 // Initialize values to zero
 int i;
 for (i = 0; i < size; i++) {
 rayIntersectionsCount[i] = 0;
 rayMissesCount[i] = 0;
 }
}

StatisticsDataStructCpp() {
 delete[] rayIntersectionsCount; rayIntersectionsCount = NULL;
 delete[] rayMissesCount; rayMissesCount = NULL;
}

};

struct StatisticsDataStructOpenCL {
 size_t size;
 // Statistics data
 cl_mem rayIntersectionsCount;
 cl_mem rayMissesCount;

StatisticsDataStructOpenCL(size_t a_Size = 0) {
 size = a_Size;
}
// These structs are meant to emulate the class-less nature of the OpenCL ray-tracer implementation

typedef struct {
    float red;
    float green;
    float blue;
} ColorS;

typedef struct {
    vector3 origin;
    vector3 direction;
    float hitDistance;
} RayS;

typedef struct {
    int type;
    int index;
} PrimitiveS;

}; // namespace Application

#include <iostream>
#include "ApplicationMain.h"
#include "scene.h"
#include "time.h"
#include "Util.h"
#include "OpenCLManager.h"
#include <vector>

namespace Application {

using namespace Common;
using namespace OpenCLRelated;

// Functions to assist in emulating OpenCL Iterative ray-tracer implementation
vector3 sphere_normal(vector3 a_PI, vector3 a_Center, float a_ReciprocalRadius)
{
    vector3 r = (a_PI - a_Center) * a_ReciprocalRadius;
    return r;
};

Ray::Ray( vector3& a_Origin, vector3& a_Dir ) :
    m_Origin( a_Origin ),
    m_Direction( a_Dir )
{
}

Engine::Engine()
{
    m_Scene = new Scene();
}

Engine::~Engine()
{
    delete m_Scene;
}

} // namespace Application
// Engine::SetTarget
// Sets the render target canvas
// -----------------------------------------------------------
void Engine::SetTarget( Pixel* a_Dest, int a_Width, int a_Height )
{
    // set pixel buffer address & size
    m_Dest = a_Dest;
    m_Width = a_Width;
    m_Height = a_Height;
}
// -----------------------------------------------------------
// Engine::Raytrace
// Naive ray tracing: Intersects the ray with every primitive
// in the scene to determine the closest intersection
// -----------------------------------------------------------
Primitive* Engine::Raytrace( Ray& a_Ray, Color& a_Acc, int a_Depth, int a_MaxDepth, float a_RIndex, float& a_Dist )
{
    if (a_Depth > a_MaxDepth) return 0;
    // trace primary ray
    a_Dist = 1000000.0f;
    vector3 pi;
    Primitive* prim = 0;
    int result;
    // ---- Find the nearest intersection ----
    for ( int s = 0; s < m_Scene->GetNrPrimitives(); s++ )
    {
        Primitive* pr = m_Scene->GetPrimitive( s );
        int res;
        if (res = pr->Intersect( a_Ray, a_Dist ))
        {
            prim = pr;
            result = res; // 0 = miss, 1 = hit, -1 = hit from inside
        }
    }
    // ---- Handle intersection ----
    if (prim) {
        // Update statistics
        m_statisticsDataStructCpp->rayIntersectionsCount[0]++;
        if (prim->IsLight()) {
            // -- Have hit a light, stop tracing --
            a_Acc = Color( 1.0f, 1.0f, 1.0f );
        }
    } else {
        // -- Determine color at point of intersection --
        pi = a_Ray.GetOrigin() + a_Ray.GetDirection() * a_Dist;
        // trace lights
        for ( int l = 0; l < m_Scene->GetNrPrimitives(); l++ )
        {
            Primitive* p = m_Scene->GetPrimitive( l );
            if (p->IsLight())
            {
                Primitive* light = p;
                // handle point light source
                float shade = 1.0f;
                if (light->GetType() == Primitive::SPHERE)
                {
                    vector3 L = ((Sphere*)light)->GetCentre() - pi;
                    float tdist = LENGTH( L );
                    NORMALIZE(L);
                    vector3 TempVector3(pi + L * EPSILON);
                    Ray r = Ray( TempVector3, L );
                    for ( int s = 0; s < m_Scene-
>GetNrPrimitives(); s++ )
    {
      Primitive* pr = m_Scene->GetPrimitive( s
    );
      if ((pr != light) && (pr->Intersect( r,
        tdist )))
      {
        shade = 0;
        break;
      }
    }
  }
  if (shade > 0)
    {
    vector3 L = ((Sphere*)light)->GetCentre() -
      pi;
    NORMALIZE( L );
    vector3 N = prim->GetNormal( pi );
    // determine diffuse component
    if (prim->GetMaterial()->GetDiffuse() > 0)
    {
      float dot = DOT( L, N );
      if (dot > 0)
      {
        float diff = dot * prim->
          >GetMaterial()->GetDiffuse() * shade;
        // add diffuse component to ray
        Color ncol = diff * prim->
          >GetMaterial()->GetColor();
        // Scale back the color, if
        necessary
        if (ncol.r > 1.0f || ncol.g >
          1.0f || ncol.b > 1.0f)
        {
          float max = 1.0f;
          if (ncol.r > max) max =
            ncol.r;
          if (ncol.g > max) max =
            ncol.g;
          if (ncol.b > max) max =
            ncol.b;
          ncol *= 1.0f/max;
        }
        a_Acc += ncol;
      }
    }
    // determine specular component
    if (prim->GetMaterial()->GetSpecular() > 0)
    {
      // point light source: sample once for
      specular highlight
      vector3 V = a_Ray.GetDirection();
      vector3 R = L - 2.0f * DOT( L, N ) * N;
      float dot = DOT( V, R );
      if (dot > 0)
      {
        float spec = powf( dot, 20 ) *
          prim->GetMaterial()->GetSpecular() * shade;
        // add specular component to ray
        Color ncol = spec * light->
          >GetMaterial()->GetColor();
        // Adjust specular component to
        be within 0 <-> 1.0
        else if (ncol.r < 0.0f) ncol.r = 0.0f;
        if (ncol.r > 1.0f) ncol.r = 1.0f;
        if (ncol.g > 1.0f) ncol.g = 1.0f;
else if (ncol.g < 0.0f) ncol.g = 0.0f;
else if (ncol.b < 0.0f) ncol.b = 0.0f;
    if (ncol.b > 1.0f) ncol.b = 1.0f;
a_Acc += ncol;
}

// Scale back the local color contribution, if necessary
if (a_Acc.r > 1.0f || a_Acc.g > 1.0f || a_Acc.b > 1.0f)
{
    float max = 1.0f;
    if (a_Acc.r > max) max = a_Acc.r;
    if (a_Acc.g > max) max = a_Acc.g;
    if (a_Acc.b > max) max = a_Acc.b;
    a_Acc *= 1.0f/max;
}

// -- Calculate reflection --
float refl = prim->GetMaterial()->GetReflection();
if ((refl > 0.0f) && (a_Depth < a_MaxDepth))
{
    vector3 N = prim->GetNormal( pi );
    vector3 R = a_Ray.GetDirection() - 2.0f * DOT(a_Ray.GetDirection(), N ) * N;
    Color rcol( 0.0f, 0.0f, 0.0f );
    float dist;
    vector3 TempVector3(pi + R * EPSILON);
    Ray TempRay(TempVector3, R);
    Raytrace( TempRay, rcol, a_Depth + 1, a_MaxDepth, a_RIndex, dist );
    a_Acc += refl * rcol;
}
else {
    // -- No intersection occured; stop tracing --
    // Update statistics
    m_statisticsDataStructCpp->rayMissesCount[0]++;
    // Do nothing

    // return pointer to primitive hit by primary ray
    return prim;
}

// -----------------------------------------------------------
// Engine::InitRender
// Initializes the renderer, by resetting the line / tile
// counters and precalculating some values
// -----------------------------------------------------------
void Engine::InitRender()
{
    // set first and last line to draw to
    m_FirstLine = 20;
    m_LastLine = m_Height - 20;
    // set first pixel of first cohort
    m_CohortPPos = m_FirstLine * m_Width;
    // set pixel buffer address of first and last pixel
    m_LastPPos = m_LastLine * m_Width;
    // screen plane in world space coordinates
    m_ScreenWorldBoundX1 = -4, m_ScreenWorldBoundX2 = 4, m_ScreenWorldBoundY1 = m_CurrScreenWorldY = 3, m_ScreenWorldBoundY2 = -3;
    // calculate deltas for interpolation
    m_WorldUnitsPerPixelX = (m_ScreenWorldBoundX2 - m_ScreenWorldBoundX1) /
m_Width;
m_WorldUnitsPerPixelY = (m_ScreenWorldBoundY2 - m_ScreenWorldBoundY1) / m_Height;
  // statistics
  m_RayMissesCount = 0;
m_RayIntersectionsCount = 0;
}

void Engine::PreRender(RenderModes mode) {
  // Prepare linearized data
  m_spheres.clear();
m_planes.clear();
  Sphere* tempSphere;
PlanePrim* tempPlane;
  int sphereCount;
  int planesCount;
  Primitive* prim;
  for (int i = 0; i < m_Scene->GetNrPrimitives(); i++) {
    prim = m_Scene->GetPrimitive(i);
    tempSphere = dynamic_cast<Sphere*>(prim);
    if (dynamic_cast<Sphere*>(prim) != 0) {
      m_spheres.push_back(tempSphere);
      continue;
    }
  tempPlane = dynamic_cast<PlanePrim*>(prim);
  if (dynamic_cast<PlanePrim*>(prim) != 0)
    m_planes.push_back(tempPlane);
  }

  // Spheres
  m_sphereDataStructOpenCL = new SphereDataStructOpenCL(m_spheres.size());
m_sphereDataStructCpp = new SphereDataStructCpp(m_spheres.size());
  for (int i = 0; i < m_spheres.size(); i++) {
    tempSphere = m_spheres[i];
    // Geometric Data
    m_sphereDataStructCpp->centerX[i] = tempSphere->GetCentre().x;
m_sphereDataStructCpp->centerY[i] = tempSphere->GetCentre().y;
m_sphereDataStructCpp->centerZ[i] = tempSphere->GetCentre().z;
m_sphereDataStructCpp->recRadius[i] = tempSphere->GetRecRadius();
m_sphereDataStructCpp->sqRadius[i] = tempSphere->GetSqRadius();

    // Material Data
m_sphereDataStructCpp->diffuse[i] = tempSphere->GetMaterial()->GetDiffuse();
m_sphereDataStructCpp->specular[i] = tempSphere->GetMaterial()->GetSpecular();
m_sphereDataStructCpp->reflection[i] = tempSphere->GetMaterial()->GetReflection();
m_sphereDataStructCpp->refraction[i] = tempSphere->GetMaterial()->GetRefraction();
m_sphereDataStructCpp->isLight[i] = tempSphere->GetMaterial()->GetLight();

    // Color Data
m_sphereDataStructCpp->red[i] = tempSphere->GetMaterial()->GetColor().r;
m_sphereDataStructCpp->green[i] = tempSphere->GetMaterial()->GetColor().g;
m_sphereDataStructCpp->blue[i] = tempSphere->GetMaterial()->GetColor().b;

    // Light?
m_sphereDataStructCpp->isLight[i] = tempSphere->GetLight();
  }

  // Planes
  m_planeDataStructOpenCL = new PlaneDataStructOpenCL(m_planes.size());
m_planeDataStructCpp = new PlaneDataStructCpp(m_planes.size());
  for (int i = 0; i < m_planes.size(); i++) {
    tempPlane = m_planes[i];
    // Geometric data
m_planeDataStructCpp->normalX[i] = tempPlane->GetNormal().x;
m_planeDataStructCpp->normalY[i] = tempPlane->GetNormal().y;
m_planeDataStructCpp->normalZ[i] = tempPlane->GetNormal().z;
m_planeDataStructCpp->d[i] = tempPlane->GetD();
// Material data
m_planeDataStructCpp->diffuse[i] = tempPlane->GetMaterial()->GetDiffuse();
m_planeDataStructCpp->specular[i] = tempPlane->GetMaterial()->GetSpecular();
m_planeDataStructCpp->reflection[i] = tempPlane->GetMaterial()->GetReflection();
m_planeDataStructCpp->refraction[i] = tempPlane->GetMaterial()->GetRefraction();
m_planeDataStructCpp->reflIndex[i] = tempPlane->GetMaterial()->GetReflIndex();
// Color data
m_planeDataStructCpp->red[i] = tempPlane->GetMaterial()->GetColor().r;
m_planeDataStructCpp->green[i] = tempPlane->GetMaterial()->GetColor().g;
m_planeDataStructCpp->blue[i] = tempPlane->GetMaterial()->GetColor().b;
// Light data
m_planeDataStructCpp->isLight[i] = tempPlane->IsLight();

// Statistics
m_RayMissesCount = 0;
m_RayIntersectionsCount = 0;

m_statisticsDataStructOpenCL = new StatisticsDataStructOpenCL((size_t)m_CohortPSize);
StatisticsDataStructCpp = new StatisticsDataStructCpp((size_t)m_CohortPSize);

if (mode == OpenCL_CPU || mode == OpenCL_GPU) {
    // Initialize OpenCL related structs
    OpenCLManager::Instance()->InitializePrimitiveDataMem(
        m_sphereDataStructCpp, m_planeDataStructCpp,
        m_sphereDataStructOpenCL, m_planeDataStructOpenCL);
    OpenCLManager::Instance()->InitializeStatisticsDataMem(
        m_statisticsDataStructCpp, m_statisticsDataStructOpenCL);
}

// Engine::Render
// Fires rays in the scene one scanline at a time, from left
// to right
// --------------------------------------------------------------------------
bool Engine::Render(RenderModes mode) {
    // render scene
    vector3 o( 0, 0, -5 );
    int j = 0;
    // initialize timer
    int msecs = Util::Time::GetTimeMilliseconds();
    if (mode == OpenCL_CPU || mode == OpenCL_GPU) {
        // render remaining lines
        for (; m_CohortPPos < m_LastPPos; m_CohortPPos += m_CohortPSize) {
            if (m_LastPPos - m_CohortPPos < m_CohortPSize) {
                m_CohortPSize = (m_LastPPos - m_CohortPPos);
            }
            // render pixels for current cohort
            // Initialize GPGPU cl_mems
            PixelDataStructOpenCL* pds_ocl = new PixelDataStructOpenCL((size_t)m_CohortPSize);
            OpenCLManager::Instance()->InitializePixelDataMem(pds_ocl);
            // GO!!
OpenCLManager::Instance()->RayTraceKernel(
    m_maxTraceDepth,
    o.x, o.y, o.z,
    m_Width, m_CohortPPos, m_ScreenWorldBoundX1,
    m_ScreenWorldBoundY1,
    m_WorldUnitsPerPixelX, m_WorldUnitsPerPixelY,
    pds_ocl, m_planeDataStructOpenCL, m_sphereDataStructOpenCL,
    m_statisticsDataStructOpenCL);

    // Read pixel data from OpenCL device memory
    OpenCLManager::Instance()->ReadPixelData(pds_ocl,
        &m_Dest[(unsigned int)m_CohortPPos]);

    // Dispose GPGPU structs
    OpenCLManager::Instance()->DisposePixelDataMem(pds_ocl);
    delete pds_ocl;

    // see if we've been working too long already
    if ((Util::Time::GetTimeMilliseconds() - msecs) > 100) {
        // return control to windows so the screen gets updated
        m_CohortPPos += m_CohortPSize;
        return false;
    }

    if (mode == GCC_CPU_Recursive) {
        // reset last found primitive pointer
        Primitive* lastprim = 0;

        // render remaining lines
        for (; m_CohortPPos < m_LastPPos; m_CohortPPos += m_CohortPSize) {
            if ((m_LastPPos - m_CohortPPos) < m_CohortPSize)
                m_CohortPSize = (m_LastPPos - m_CohortPPos);
            for (j = m_CohortPPos; j < (m_CohortPPos + m_CohortPSize); j++) {
                // calculate screen world x and y
                m_CurrScreenWorldX = j % m_Width * m_WorldUnitsPerPixelX + m_ScreenWorldBoundX1;
                m_CurrScreenWorldY = j / m_Width * m_WorldUnitsPerPixelY + m_ScreenWorldBoundY1;

                // fire primary rays
                Color acc(0.0f, 0.0f, 0.0f);
                vector3 dir = vector3(m_CurrScreenWorldX, m_CurrScreenWorldY, 0) - o;
                NORMALIZE(dir);
                Ray r(o, dir);
                float dist;
                Primitive* prim = Raytrace(r, acc, 1, m_maxTraceDepth, 1.0f, dist);

                int red, green, blue;
                red = (int)(acc.r * 256);
                green = (int)(acc.g * 256);
                blue = (int)(acc.b * 256);

                if (red > 255) red = 255;
                if (green > 255) green = 255;
                if (blue > 255) blue = 255;

                m_Dest[j] = (blue << 16) + (green << 8) + red;

            }

            // see if we've been working to long already
            if ((Util::Time::GetTimeMilliseconds() - msecs) > 100) {
                // return control to windows so the screen gets updated
                m_CohortPPos += m_CohortPSize;
                return false;
            }
        }
    }
if (mode == GCC_CPU_Iterative) {
    for (; m_CohortPPos < m_LastPPos; m_CohortPPos += m_CohortPSize) {
        if ((m_LastPPos - m_CohortPPos) < m_CohortPSize) {
            m_CohortPSize = (m_LastPPos - m_CohortPPos);
        }
        for (j = m_CohortPPos; j < (m_CohortPPos + m_CohortPSize); j++) {
            // USEFUL STUFF
            // RayCast stuff
            vector3 camPos3 = 0;
            vector3 screenCoord3{
                j % m_Width * m_WorldUnitsPerPixelX + m_ScreenWorldBoundX1,
                j / m_Width * m_WorldUnitsPerPixelY + m_ScreenWorldBoundY1,
            }
            RayS rayCast;
            rayCast.origin = camPos3;
            rayCast.direction = screenCoord3 - camPos3;
            NORMALIZE(rayCast.direction);
            // Last-primitive-intersected data stuff
            PrimitiveS lastPrim;
            vector3 pi3;
            // Color components stuff
            ColorS color = { 0.0f, 0.0f, 0.0f };  // Hard coded to
            // Reflection coefficients
            float reflectionCoeffs[17];
            maxDepth = 16
            reflectionCoeffs[0] = 1.0f;
            // Misc stuff
            bool done = false;
            int currentDepth = 1;
            // For each level of the ray trace tree
            while ( (currentDepth <= m_maxTraceDepth) && (!done) ) {
                // Create temporary color
                ColorS localColor = { 0.0f, 0.0f, 0.0f };  // Hard coded to
                // Reset some values
                rayCast.hitDistance = 1000000.0f;
                lastPrim.type = NO_HIT;
                lastPrim.index = -1;
                // ---- Find the nearest intersection ----
                for (int p = 0; p < m_planeDataStructCpp->size; p++) {
                    // Intersect with planes
                    vector3 n3 = vector3(m_planeDataStructCpp->normalX[p],
                                          m_planeDataStructCpp->normalY[p],
                                          m_planeDataStructCpp->normalZ[p]);
                    float d = DOT(n3, rayCast.direction);
                    if (d != 0) {
                        float dist = -(DOT(n3, rayCast.origin) +
                                       m_planeDataStructCpp->d[p]) / d;
                        if (dist > 0) {
                            if (dist < rayCast.hitDistance) {
                                rayCast.hitDistance = dist;
                                lastPrim.type = PLANE_HIT;
                                lastPrim.index = p;
                            }
                        }
                    }
                }
                for (int s = 0; s < m_sphereDataStructCpp->size; s++) {
                    // Intersect with spheres
                }
            }
        }
    }
}
vector3 c3 = vector3(m_sphereDataStructCpp->centerX[s], m_sphereDataStructCpp->centerY[s],
                m_sphereDataStructCpp->centerZ[s]);
vector3 v3 = rayCast.origin - c3;
float b = -DOT(v3, rayCast.direction);
float det = (b * b) - DOT(v3, v3) +
            m_sphereDataStructCpp->sqRadius[s];
if (det > 0) {
    det = sqrt(det);
    float i1 = b - det;
    float i2 = b + det;
    if (i2 > 0) {
        if (i1 < 0) {
            if (i2 <
                rayCast.hitDistance) {
                rayCast.hitDistance = i2;
                lastPrim.type = SPHERE_HIT_INSIDE;
                lastPrim.index = s;
            }
        }
    }
}
else {
    if (i1 <
        rayCast.hitDistance) {
        rayCast.hitDistance = i1;
        lastPrim.type = SPHERE_HIT_OUTSIDE;
        lastPrim.index = s;
    }
}

// ---- Handle intersection ----
if (lastPrim.type != NO_HIT) {
    // Update statistics
    m_statisticsDataStructCpp->rayIntersectionsCount[0]++;
    if ( (lastPrim.type == SPHERE_HIT_OUTSIDE ||
            lastPrim.type == SPHERE_HIT_INSIDE) &&
        m_sphereDataStructCpp->isLight[lastPrim.index] == 1) {
        // -- Have hit a light, stop tracing
        localColor.red = 1.0f; localColor.green = 1.0f;
        done = true;
    } else {
        // -- Determine color at point of intersection --
        pi3 = rayCast.origin +
            (rayCast.direction * rayCast.hitDistance);
            // trace lights
            for (int l = 0; l <
                m_sphereDataStructCpp->size; l++) {
                if (m_sphereDataStructCpp->isLight[l] == 1) // If it is a light
                { // handle point light
                    source = vector3 lc3 =
                }
            }
    }
}

vector3(m_sphereDataStructCpp->centerX[l],
         m_sphereDataStructCpp->centerY[l],
         m_sphereDataStructCpp->centerZ[l]);

float shade = 1.0f;
vector3 L3 = lc3 - pi3;
float tdist = LENGTH(L3);
NORMALIZE(L3);
// Create light ray
RayS lightRay;
lightRay.origin = pi3 + L3;
lightRay.direction = L3;
if (shade > 0)
{
    // Check for hit against planes
    for (int p = 0; p < m_planes.size(); p++ )
    {
        vector3 n3 =
            vector3(m_planeDataStructCpp->normalX[p],
                    m_planeDataStructCpp->normalY[p],
                    m_planeDataStructCpp->normalZ[p]);
        float d = DOT(n3, lightRay.direction);
        if (d != 0)
        {
            float dist = -(DOT(n3, lightRay.origin) + m_planeDataStructCpp->d[p]) / d;
            if (dist > 0)
            {
                if (dist < tdist)
                {
                    tdist = dist;
                    shade = 0;
                    break;
                }
            }
        }
    }
}
if (shade > 0) // If there has not already been a plane hit
{
    // Check for hit against spheres
    for (int s = 0; s < m_sphereDataStructCpp->size; s++)
    {
        if (m_sphereDataStructCpp->isLight[s] != 0)
            continue;
        vector3 sc3 =
            vector3(m_sphereDataStructCpp->centerX[s],
                    m_sphereDataStructCpp->centerY[s],
                    m_sphereDataStructCpp->centerZ[s]);
        }
m_sphereDataStructCcpp->centerZ[s]);

lightRay.origin - sc3;
DOT(v3, lightRay.direction);
(b * b) - DOT(v3, v3) + m_sphereDataStructCcpp->sqRadius[s];

if (det > 0)
{
    det = sqrt(det);
    i1 = b - det;
    i2 = b + det;
    if (i2 > 0) {
        if (i1 < 0) {
            if (i2 < tdist) {
                tdist = i2;
                shade = 0;
                break;
            }
        } else {
            if (i1 < tdist) {
                tdist = i1;
                shade = 0;
                break;
            }
        }
    } else {
        if (i2 < tdist) {
            if (i2 < tdist) {
                tdist = i2;
                shade = 0;
                break;
            }
        } else {
            if (i1 < tdist) {
                tdist = i1;
                shade = 0;
                break;
            }
        }
    }
}

from light.

if (shade > 0) {
    vector3 lc3 =
    m_sphereDataStructCcpp->centerX[l],
    m_sphereDataStructCcpp->centerY[l],
    m_sphereDataStructCcpp->centerZ[l]);

    vector3 L3 = lc3 -
    pi3;
    NORMALIZE(L3);
    vector3 n3;
    // If last prim was
    if (lastPrim.type ==
PLANE_HIT) {  
    vector3 n3 = vector3(m_planeDataStructCpp->normalX[lastPrim.index],  
        m_planeDataStructCpp->normalY[lastPrim.index],  
        m_planeDataStructCpp->normalZ[lastPrim.index]);  
    // calculate DIFFUSE SHADING  
    if (m_planeDataStructCpp->diffuse[lastPrim.index] > 0) {  
        float dot = DOT(L3, n3);  
        if (dot > 0) {  
            float diff = dot * m_planeDataStructCpp->diffuse[lastPrim.index];  
            ColorS ncol = { diff * m_planeDataStructCpp->red[lastPrim.index],  
                diff * m_planeDataStructCpp->green[lastPrim.index],  
                diff * m_planeDataStructCpp->blue[lastPrim.index] };  
            // Scale back the color, if necessary  
            if (ncol.red > 1.0f || ncol.green > 1.0f || ncol.blue > 1.0f)  
                {  
            float max = 1.0f;  
            if (ncol.red > max) max = ncol.red;  
            if (ncol.blue > max) max = ncol.blue;  
            if (ncol.green > max) max = ncol.green;  
                ncol.red *= 1.0f/max;  
                ncol.green *= 1.0f/max;  
                ncol.blue *= 1.0f/max;  
            }  
            localColor.red += ncol.red;  
            localColor.green += ncol.green;  
            localColor.blue += ncol.blue;  
        }  
    }  
    // determine SPECULAR COMPONENT  
    if (m_planeDataStructCpp->specular[lastPrim.index] > 0) {  
        // point light source: sample once for specular highlight  
        vector3 v3 = rayCast.direction;  
        vector3 R3 = L3 - n3*(2.0f*DOT(L3, n3));  
        float dot = DOT(v3, R3);  
        if (dot > 0) {  
            float specular = m_planeDataStructCpp->specular[lastPrim.index] * shade;  
            ColorS ncol = { specular * m_planeDataStructCpp->red[lastPrim.index],  
                specular * m_planeDataStructCpp->green[lastPrim.index],  
                specular * m_planeDataStructCpp->blue[lastPrim.index] };  
            // Scale back the color, if necessary  
            if (ncol.red > 1.0f || ncol.green > 1.0f || ncol.blue > 1.0f)  
                {  
            float max = 1.0f;  
            if (ncol.red > max) max = ncol.red;  
            if (ncol.blue > max) max = ncol.blue;  
            if (ncol.green > max) max = ncol.green;  
                ncol.red *= 1.0f/max;  
                ncol.green *= 1.0f/max;  
                ncol.blue *= 1.0f/max;  
            }  
            localColor.red += ncol.red;  
            localColor.green += ncol.green;  
            localColor.blue += ncol.blue;  
        }  
    }
(dot > 0) {
    float spec = pow(dot, 20) * m_planeDataStructCpp->specular[lastPrim.index] * shade;
    // add specular component to ray color
    ColorS ncol = { spec * m_sphereDataStructCpp->red[l],
                   spec * m_sphereDataStructCpp->green[l],
                   spec * m_sphereDataStructCpp->blue[l] };
    // Adjust specular component to be within 0 <-> 1.0
    if (ncol.red > 1.0f) ncol.red = 1.0f; else if (ncol.red < 0.0f) ncol.red = 0.0f;
    if (ncol.green > 1.0f) ncol.green = 1.0f; else if (ncol.green < 0.0f) ncol.green = 0.0f;
    if (ncol.blue > 1.0f) ncol.blue = 1.0f; else if (ncol.blue < 0.0f) ncol.blue = 0.0f;
    localColor.red += ncol.red;
    localColor.green += ncol.green;
    localColor.blue += ncol.blue;
}

// If last prim was a sphere...
if (lastPrim.type == SPHERE_HIT_INSIDE || lastPrim.type == SPHERE_HIT_OUTSIDE) {
    float recRadius = m_sphereDataStructCpp->recRadius[lastPrim.index];
    vector3 sphereCenter3 = vector3(m_sphereDataStructCpp->centerX[lastPrim.index],
                                    m_sphereDataStructCpp->centerY[lastPrim.index],
                                    m_sphereDataStructCpp->centerZ[lastPrim.index]);
    n3 = sphere_normal(pi3, sphereCenter3, recRadius);
    // calculate DIFFUSE SHADING
    (m_sphereDataStructCpp->diffuse[lastPrim.index] > 0) {
        float dot = DOT (L3, n3);
        if (dot > 0) {
            float diff = dot * m_sphereDataStructCpp->diffuse[lastPrim.index] * shade;
            // add diffuse component to ray color
            ColorS ncol = { diff * m_sphereDataStructCpp->red[lastPrim.index],
                           diff * m_sphereDataStructCpp->green[lastPrim.index],
                           diff * m_sphereDataStructCpp->blue[lastPrim.index] };
            // Scale back the color, if necessary
if (ncol.red > 1.0f || ncol.green > 1.0f || ncol.blue > 1.0f)
{
  float max = 1.0f;
  if (ncol.red > max) max = ncol.red;
  if (ncol.green > max) max = ncol.green;
  if (ncol.blue > max) max = ncol.blue;
  ncol.red *= 1.0f/max;
  ncol.green *= 1.0f/max;
  ncol.blue *= 1.0f/max;
}
localColor.red += ncol.red;
localColor.green += ncol.green;
localColor.blue += ncol.blue;

// determine SPECULAR COMPONENT
if (m_sphereDataStructCpp->specular[lastPrim.index] > 0) {
  //point light source: sample once for specular highlight
  vector3 v3 = rayCast.direction;
  vector3 r3 = L3 - (n3*2.0f*DOT(L3, n3));
  float dot = DOT(v3, r3);
  if (dot > 0) {
    float spec = pow(dot, 20) * m_sphereDataStructCpp->specular[lastPrim.index] * shade;
    // add specular component to ray color
    ColorS ncol = { spec * m_sphereDataStructCpp->red[1],
                   spec * m_sphereDataStructCpp->green[1],
                   spec * m_sphereDataStructCpp->blue[1] };
    // Adjust specular component to be within 0 <-> 1.0
    if (ncol.red > 1.0f) ncol.red = 1.0f; else if (ncol.red < 0.0f) ncol.red = 0.0f;
    if (ncol.green > 1.0f) ncol.green = 1.0f; else if (ncol.green < 0.0f) ncol.green = 0.0f;
    if (ncol.blue > 1.0f) ncol.blue = 1.0f; else if (ncol.blue < 0.0f) ncol.blue = 0.0f;
    localColor.red += ncol.red;
    localColor.green += ncol.green;
localColor.blue += ncol.blue;
}

// -- Calculate reflection --
float refl;
if (lastPrim.type == SPHERE_HIT_INSIDE || lastPrim.type == SPHERE_HIT_OUTSIDE) {
    refl = m_sphereDataStructCpp->reflection[lastPrim.index];
}
if (lastPrim.type == PLANE_HIT) {
    refl = m_planeDataStructCpp->reflection[lastPrim.index];
}
if (refl > 0.0f) {
    vector3 n3;
    if (lastPrim.type == SPHERE_HIT_INSIDE || lastPrim.type == SPHERE_HIT_OUTSIDE) {
        float recRadius = m_sphereDataStructCpp->recRadius[lastPrim.index];
        vector3 sphereCenter3 = vector3(m_sphereDataStructCpp->centerX[lastPrim.index],
                                          m_sphereDataStructCpp->centerY[lastPrim.index],
                                          m_sphereDataStructCpp->centerZ[lastPrim.index]);

        float recrad = recRadius;
        vector3 n3 = sphere_normal(pi3, sphereCenter3, recRadius);
    }
    if (lastPrim.type == PLANE_HIT) {
        n3 = vector3(m_planeDataStructCpp->normalX[lastPrim.index],
                     m_planeDataStructCpp->normalY[lastPrim.index],
                     m_planeDataStructCpp->normalZ[lastPrim.index]);
    }
    vector3 r3 = rayCast.direction - (DOT(rayCast.direction, n3) * n3 * 2.0f);
    rayCast.origin = r3 * EPSILON + pi3;
    refl1;
}
else {
    // -- No more reflections needed
    done = true;
}
}
else {
    // Update statistics
    m_statisticsDataStructCpp->rayMissesCount[0]++;
    // -- No intersection occured; stop tracing --
    done = true;
}
// ---- Adjust contribution of reflected ray ----
// Scale back the local color contribution, if necessary
if (localColor.red > 1.0f || localColor.green > 1.0f || localColor.blue > 1.0f)
{
    float max = 1.0f;
    if (localColor.red > max) max = localColor.red;
    if (localColor.green > max) max = localColor.green;
    if (localColor.blue > max) max = localColor.blue;
    localColor.red *= 1.0f/max;
    localColor.green *= 1.0f/max;
    localColor.blue *= 1.0f/max;
}

// -- Weight the reflection coefficient --
float rc = reflectionCoeffs[0];
for (int i = 1; i < currentDepth; i++) {
    rc *= reflectionCoeffs[i];
}
localColor.red *= rc;
localColor.green *= rc;
localColor.blue *= rc;

// update color
color.red += localColor.red;
color.green += localColor.green;
color.blue += localColor.blue;
++currentDepth;
// end while loop

// Process color values
int r, g, b;
r = (int)(color.red * 256);
g = (int)(color.green * 256);
b = (int)(color.blue * 256);
if (r > 255) r = 255;
if (g > 255) g = 255;
if (b > 255) b = 255;
m_Dest[j] = (b << 16) + (g << 8) + r;

// see if we've been working too long already
if ((Util::Time::GetTimeMilliseconds() - msecs) > 100)
{
    // return control to windows so the screen gets updated
    m_CohortPPos += m_CohortPSize;
    return false;
}

} // end GCC_CPU_Iterative

// all done
return true;

void Engine::PostRender(RenderModes mode)
{
    if (mode == OpenCL_CPU || mode == OpenCL_GPU) {
// Retrieve OpenCL Kernel's statistics data
OpenCLManager::Instance()->ReadStatisticsData(m_statisticsDataStructOpenCL,

m_statisticsDataStructCpp);
    // Dispose of OpenCL related structs
    OpenCLManager::Instance()->
      DisposePrimitiveDataMem(m_planeDataStructOpenCL,
      m_sphereDataStructOpenCL);
    m_sphereDataStructOpenCL);
    OpenCLManager::Instance()->
      DisposeStatisticsDataMem(m_statisticsDataStructOpenCL);
    }
    // Update statistics
    for (int i = 0; i < m_statisticsDataStructCpp->size; i++) {
      m_RayIntersectionsCount += m_statisticsDataStructCpp-
        rayIntersectionsCount[i];
      m_RayMissesCount += m_statisticsDataStructCpp->rayMissesCount[i];
    }
    delete m_sphereDataStructCpp;
    delete m_sphereDataStructOpenCL;
    delete m_planeDataStructCpp;
    delete m_planeDataStructOpenCL;
    delete m_statisticsDataStructCpp;
    delete m_statisticsDataStructOpenCL;
  }

int Engine::GetRayIntersectionsCount() {
  return m_RayIntersectionsCount;
}

int Engine::GetRayMissesCount() {
  return m_RayMissesCount;
}

void Engine::SetMaxTraceDepth(int a_MaxDepth) {
  m_maxTraceDepth = a_MaxDepth;
}

void Engine::SetCohortSize(int a_CohortPSize) {
  m_CohortPSize = a_CohortPSize;
}

void Engine::SetWorkgroupSize(int a_WorkgroupSize) {
  OpenCLManager::Instance()->
    SetRaytraceKernelWorkGroupSize(a_WorkgroupSize);
}
}; // namespace Application

#endif // APPLICATION_MAIN_H
#define APPLICATION_MAIN_H
#include "common.h"
#include "app_common.h"
#include "scene.h"
namespace Application {
  // -----------------------------------------------
  // ApplicationMain.h
  // 2011 - modified - Gary Deng - dengsta@gmail.com
  // 2004 - original - Jacco Bikker - jacco@bik5.com - www.bik5.com - <><
  // Application's Main class
  // -----------------------------------------------

  #ifndef I_APPLICATION_MAIN_H
  #define I_APPLICATION_MAIN_H

  #include "common.h"
  #include "app_common.h"
  #include "scene.h"

  namespace Application {
    // -----------------------------------------------
    // Class prototypes
    // -----------------------------------------------
    class Sphere;
class PlanePrim;

// Ray class definition
class Ray {
public:
  Ray() : m_Origin( vector3( 0, 0, 0 ) ), m_Direction( vector3( 0, 0, 0 ) ) {};
  Ray( vector3& a_Origin, vector3& a_Dir );
  void SetOrigin( vector3& a_Origin ) { m_Origin = a_Origin; }
  void SetDirection( vector3& a_Direction ) { m_Direction = a_Direction; }
  vector3& GetOrigin() { return m_Origin; }
  vector3& GetDirection() { return m_Direction; }
private:
  vector3 m_Origin;
  vector3 m_Direction;
};
StatisticsDataStructOpenCL* m_statisticsDataStructOpenCL;
};

}; // namespace Application

#endif

// -----------------------------------------------------------
// common.h
// 2011 - Gary Deng - dengsta@gmail.com
// Common application definitions
// -----------------------------------------------------------

#ifndef I_ROOT_COMMON_H
#define I_ROOT_COMMON_H

#include "app_common.h"

namespace Common {

#define OGL_SCRWIDTH800
#define OGL_SCRHEIGHT 600

} // namespace Common

#endif

// -----------------------------------------------------------
// gui_common.h
// 2011 - Gary Deng - dengsta@gmail.com
// Common GUI definitions
// -----------------------------------------------------------

#ifndef I_GUI_COMMON_H
#define I_GUI_COMMON_H

#include "app_common.h"

namespace GUI {

class RenderMode {
public:
    RenderMode(Raytracer::RenderModes startingMode = Raytracer::GCC_CPU) {
        currentMode = startingMode;
    }

    ~RenderMode() {
    }

    Raytracer::RenderModes GetCurrentMode() {
        return currentMode;
    }

    Raytracer::RenderModes SetCurrentMode(Raytracer::RenderModes a_Mode) {
        currentMode = a_Mode;
    }

private:
    Raytracer::RenderModes currentMode;
};

} // namespace GUI

#endif

/*
==============================================================================
*/
This file is part of the JUCE library - "Jules' Utility Class Extensions"
Copyright 2004-9 by Raw Material Software Ltd.

JUCE can be redistributed and/or modified under the terms of the GNU General
Public License (Version 2), as published by the Free Software Foundation.
A copy of the license is included in the JUCE distribution, or can be found
online at www.gnu.org/licenses.

JUCE is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
A PARTICULAR PURPOSE. See the GNU General Public License for more details.

To release a closed-source product which uses JUCE, commercial licenses are
available: visit www.rawmaterialsoftware.com/juce for more information.

// gui_headers.h
// 2011 - Gary Deng - dengsta@gmail.com
// JUCE's common definitions

#ifndef __HEADERS_JUCEHEADER__
#define __HEADERS_JUCEHEADER__

// Include the JUCE headers..
#include "../JuceLibraryCode/JuceHeader.h"
// Include local headers..
#include "MainComponent.h"

#if JUCE_IPHONE || JUCE_LINUX
#undef JUCE_USE_CAMERA
#endif

// Pre-declare the functions that create each of the components..
#if JUCE_OPENGL
    Component* createOpenGLWindow();
#endif

#endif  // __HEADERS_JUCEHEADER__

/*
This file was auto-generated by the Jucer!

It contains the basic startup code for a Juce application.

==============================================================================

*/

// GuiMain.cpp
// 2011 - Gary Deng - dengsta@gmail.com
// JUCE's GUI's main class

#include "../JuceLibraryCode/JuceHeader.h"
#include "gui_headers.h"
This is the top-level window that we'll pop up. Inside it, we'll create and show a component from the MainComponent.cpp file (you can open this file using the Jucer to edit it).

```cpp
class RayTracerComparisonGUIWindow : public DocumentWindow
{
public:

RayTracerComparisonGUIWindow()
: DocumentWindow (T("Ray Tracer Comparison"),
    Colours::lightgrey,
    DocumentWindow::allButtons,
    true)
{
    // Create an instance of our main content component, and add it
    // to our window.
    MainComponent* const contentComponent = new MainComponent();
    setContentComponent (contentComponent, true, true);
    centreWithSize (getWidth(), getHeight());
    setVisible (true);
}

~RayTracerComparisonGUIWindow()
{
    // (the content component will be deleted automatically, so no need to
do it here)
}

void closeButtonPressed()
{
    // When the user presses the close button, we'll tell the app to quit.
    This
    // window will be deleted by our HelloWorldApplication::shutdown()
    // method
    //
    // JUCEApplication::quit();
}
};

// This is the application object that is started up when Juce starts. It
// handles
// the initialisation and shutdown of the whole application.

class RayTracerComparisonGUIApplication : public JUCEApplication
{
    RayTracerComparisonGUIWindow* rayTracerComparisonGUIWindow;
public:

RayTracerComparisonGUIApplication()
{
    // Don't do anything in this constructor! It will be called before the
    // main Juce subsystem has been initialised!
}

~RayTracerComparisonGUIApplication()
{
    // Don't do anything in this destructor! It will be called after the
```
void initialise (const String& commandLine)
{
    // Do your application's initialisation code here..
    rayTracerComparisonGUIWindow = new RayTracerComparisonGUIWindow();
}

void shutdown()
{
    // Do your application's shutdown code here..
    if (rayTracerComparisonGUIWindow != 0)
    {
        delete rayTracerComparisonGUIWindow;
    }
}

void systemRequestedQuit()
{
    quit();
}

const String getApplicationName()
{
    return "RayTracerComparisonGUI";
}

const String getApplicationVersion()
{
    return ProjectInfo::versionString;
}

bool moreThanOneInstanceAllowed()
{
    return false;
}

void anotherInstanceStarted (const String& commandLine)
{
}

private:
};

// This macro generates the main() routine that starts the app.
START_JUCE_APPLICATION(RayTracerComparisonGUIApplication)
/*
This is an automatically generated file created by the Jucer!

Creation date:  15 Dec 2010 9:46:10pm

Be careful when adding custom code to these files, as only the code within
the "//[xyz]" and "//[/xyz]" sections will be retained when the file is loaded
and re-saved.

Jucer version: 1.12
The Jucer is part of the JUCE library - "Jules' Utility Class Extensions"
Copyright 2004-6 by Raw Material Software ltd.

--------
// MainComponent.cpp
// 2011 - Gary Deng - dengsta@gmail.com
// The viewing window

// [Headers] You can add your own extra header files here...
#include "MainComponent.h"
#include "OpenGLWindow.h"
#include <assert.h>

// [MiscUserDefs] You can add your own user definitions and misc code here...
using namespace Common;

>MainComponent::MainComponent ()
    : ImplementationLabel (0),
      ImplementationCombo (0),
      PixelCohortSizeLabel (0),
      PixelCohortSizeComboBox (0),
      WorkGroupSizeLabel (0),
      WorkGroupSizeLabel2 (0),
      TraceDepthLabel (0),
      TraceDepthSlider (0),
      RenderButton (0),
      OGLComponent (0),
      RayIntersectionNumberLabel (0),
      RayIntersectionNumberLabel2 (0),
      RayMissNumberLabel (0),
      RayMissNumberLabel2 (0)
{
    addAndMakeVisible (ImplementationLabel = new Label
        (T("ImplementationLabel"),
        T("Implementation" : ")));
    ImplementationLabel->setFont (Font (15.0000f, Font::plain));
    ImplementationLabel->setJustificationType (Justification::centredLeft);
    ImplementationLabel->setEditable (false, false, false);
    ImplementationLabel->setColour (TextEditor::textColourId, Colours::black);
    ImplementationLabel->setColour (TextEditor::backgroundColourId, Colour (0x0));

    addAndMakeVisible (ImplementationCombo = new ComboBox
        (T("ImplementationCombo"),
        T("ImplementationComboBox")));
    ImplementationCombo->setEditableText (false);
    ImplementationCombo->setJustificationType (Justification::centredLeft);
    ImplementationCombo->setTextWhenNothingSelected (String::empty);
    ImplementationCombo->setTextWhenNoChoicesAvailable (T("(no choices)"));
    ImplementationCombo->addListener (this);

    addAndMakeVisible (PixelCohortSizeLabel = new Label
        (T("PixelCohortSizeLabel"),
        T("Pixel Cohort Size : ")));
    PixelCohortSizeLabel->setFont (Font (15.0000f, Font::plain));
    PixelCohortSizeLabel->setJustificationType (Justification::centredLeft);
    PixelCohortSizeLabel->setEditable (false, false, false);
    PixelCohortSizeLabel->setColour (TextEditor::textColourId, Colours::black);
PixelCohortSizeLabel->setColour (TextEditor::backgroundColourId, Colour (0x0));

addAndMakeVisible (PixelCohortSizeCombo = new ComboBox (T("PixelCohortSizeCombo")));
PixelCohortSizeCombo->setEditableText (false);
PixelCohortSizeCombo->setJustificationType (Justification::centredLeft);
PixelCohortSizeCombo->setTextWhenNothingSelected (String::empty);
PixelCohortSizeCombo->setTextWhenNoChoicesAvailable (T("(no choices)"));
PixelCohortSizeCombo->addListener (this);

addAndMakeVisible (WorkGroupSizeLabel = new Label (T("WorkGroupSizeLabel"),
T("OpenCL Recommended WorkGroup Size :")));
WorkGroupSizeLabel->setFont (Font (15.0000f, Font::plain));
WorkGroupSizeLabel->setJustificationType (Justification::centredLeft);
WorkGroupSizeLabel->setEditable (false, false, false);
WorkGroupSizeLabel->setColour (TextEditor::textColourId, Colours::black);
WorkGroupSizeLabel->setColour (TextEditor::backgroundColourId, Colour (0x0));

addAndMakeVisible (WorkGroupSizeLabel2 = new Label (T("WorkGroupSizeLabel2"),
T("=")));
WorkGroupSizeLabel2->setFont (Font (15.0000f, Font::plain));
WorkGroupSizeLabel2->setJustificationType (Justification::centredLeft);
WorkGroupSizeLabel2->setEditable (false, false, false);
WorkGroupSizeLabel2->setColour (TextEditor::textColourId, Colours::black);
WorkGroupSizeLabel2->setColour (TextEditor::backgroundColourId, Colour (0x0));

addAndMakeVisible (TraceDepthLabel = new Label (T("TraceDepthLabel"),
T("Max Trace Depth :")));
TraceDepthLabel->setFont (Font (15.0000f, Font::plain));
TraceDepthLabel->setJustificationType (Justification::centredLeft);
TraceDepthLabel->setEditable (false, false, false);
TraceDepthLabel->setColour (TextEditor::textColourId, Colours::black);
TraceDepthLabel->setColour (TextEditor::backgroundColourId, Colour (0x0));

addAndMakeVisible (TraceDepthSlider = new Slider (String::empty));
TraceDepthSlider->setRange (0, 16, 1.0);
TraceDepthSlider->setSliderStyle (Slider::LinearHorizontal);
TraceDepthSlider->setTextBoxStyle (Slider::TextBoxLeft, false, 70, 20);
TraceDepthSlider->addListener (this);

addAndMakeVisible (RenderButton = new TextButton (String::empty));
RenderButton->setButtonText("Render");
RenderButton->addListener (this);

OGLComponent = createOpenGLWindow();
addAndMakeVisible (OGLComponent);
OGLComponent->setName (T("OGLComponent"));

addAndMakeVisible (RayIntersectionNumberLabel = new Label (T("RayIntersectionNumberLabel"),
T("Ray-Surface Intersections :")));
RayIntersectionNumberLabel->setFont (Font (15.0000f, Font::plain));
RayIntersectionNumberLabel->setJustificationType (Justification::centredLeft);
RayIntersectionNumberLabel->setEditable (false, false, false);
RayIntersectionNumberLabel->setColour (TextEditor::textColourId, Colours::black);
RayIntersectionNumberLabel->setColour (TextEditor::backgroundColourId, Colour (0x0));
addAndMakeVisible (RayIntersectionNumberLabel2 = new Label
(T("RayIntersectionNumberLabel2"),
    T("-")));
RayIntersectionNumberLabel2->setFont (Font (15.0000f, Font::plain));
RayIntersectionNumberLabel2->setJustificationType
(Justification::centredLeft);
RayIntersectionNumberLabel2->setEditable (false, false, false);
RayIntersectionNumberLabel2->setColour (TextEditor::textColourId, Colours::black);
RayIntersectionNumberLabel2->setColour (TextEditor::backgroundColourId, Colour (0x0));

addAndMakeVisible (RayMissNumberLabel = new Label
(T("RayMissNumberLabel"),
    T("Ray Misses :")));
RayMissNumberLabel->setFont (Font (15.0000f, Font::plain));
RayMissNumberLabel->setJustificationType (Justification::centredLeft);
RayMissNumberLabel->setEditable (false, false, false);
RayMissNumberLabel->setColour (TextEditor::textColourId, Colours::black);
RayMissNumberLabel->setColour (TextEditor::backgroundColourId, Colour (0x0));

addAndMakeVisible (RayMissNumberLabel2 = new Label
(T("RayMissNumberLabel2"),
    T("-")));
RayMissNumberLabel2->setFont (Font (15.0000f, Font::plain));
RayMissNumberLabel2->setJustificationType (Justification::centredLeft);
RayMissNumberLabel2->setEditable (false, false, false);
RayMissNumberLabel2->setColour (TextEditor::textColourId, Colours::black);
RayMissNumberLabel2->setColour (TextEditor::backgroundColourId, Colour (0x0));

//[UserPreSize]
//[/UserPreSize]
setSize(OGL_SCRWIDTH + 50, OGL_SCRHEIGHT + 155);

//[Constructor] You can add your own custom stuff here..
ImplementationCombo->addItem(T("GCC CPU - Recursive"), 1);
ImplementationCombo->addItem(T("GCC CPU - Iterative"), 2);
ImplementationCombo->addItem(T("OpenCL CPU"), 3);
ImplementationCombo->addItem(T("OpenCL GPU"), 4);
ImplementationCombo->setSelectedId(1, false);
TraceDepthSlider->setValue(4);
UpdatePixelCohortSizeCombo();
//[/Constructor]

MainComponent::~MainComponent()
{
    //[Destructor_pre]. You can add your own custom destruction code here..
    //[/Destructor_pre]
    deleteAndZero (ImplementationLabel);
    deleteAndZero (ImplementationCombo);
    deleteAndZero (PixelCohortSizeLabel);
    deleteAndZero (PixelCohortSizeCombo);
    deleteAndZero (WorkGroupSizeLabel);
    deleteAndZero (WorkGroupSizeLabel2);
    deleteAndZero (TraceDepthLabel);
    deleteAndZero (TraceDepthSlider);
    deleteAndZero (RenderButton);
    deleteAndZero (OGLComponent);
deleteAndZero (RayIntersectionNumberLabel);
deleteAndZero (RayIntersectionNumberLabel2);
deleteAndZero (RayMissNumberLabel);
deleteAndZero (RayMissNumberLabel2);

// [Destructor]. You can add your own custom destruction code here..
//===]/Destructor} 

//==============================================================================

void MainComponent::paint (Graphics& g)
{
    // [UserPrePaint] Add your own custom painting code here..
    // [UserPrePaint]
    g.fillAll (Colour (0xffdbdbdb));
    // [UserPaint] Add your own custom painting code here..
    // [UserPaint] 
}

void MainComponent::resized()
{
    ImplementationLabel->setBounds (16, 24, 150, 24);
    ImplementationCombo->setBounds (162, 24, 192, 24);
    PixelCohortSizeLabel->setBounds (500, 60, 150, 24);
    PixelCohortSizeCombo->setBounds (635, 60, 128, 24);
    WorkGroupSizeLabel->setBounds (16, 24, 260, 24);
    WorkGroupSizeLabel2->setBounds (760, 24, 128, 24);
    TraceDepthLabel->setBounds (16, 60, 150, 24);
    TraceDepthSlider->setBounds (162, 60, 150, 24);
    RenderButton->setBounds (330, 60, 150, 24);
    OGLComponent->setBounds (15, 100, 800, 600);
    RayIntersectionNumberLabel->setBounds (66, 720, 200, 24);
    RayIntersectionNumberLabel2->setBounds (270, 720, 100, 24);
    RayMissNumberLabel->setBounds (500, 720, 200, 24);
    RayMissNumberLabel2->setBounds (704, 720, 100, 24);
    // [UserResized] Add your own custom resize handling here..
    // [UserResized] 
}

void MainComponent::comboBoxChanged (ComboBox* comboBoxThatHasChanged)
{
    // [UserComboBoxChanged_Pre]
    // [UserComboBoxChanged_Pre]
    if (comboBoxThatHasChanged == ImplementationCombo)
    {
        OpenGLWindow* OGLDemo;
        assert (OGLDemo = dynamic_cast<OpenGLWindow*> (OGLComponent));

        switch (comboBoxThatHasChanged->getSelectedId())
        {
        case 1:
            OGLDemo->setRenderMode(Application::GCC_CPU_Recursive);
            UpdateWorkGroupSizeLabels();
            UpdatePixelCohortSizeCombo();
            break;
        case 2:
            OGLDemo->setRenderMode(Application::GCC_CPU_Iterative);
            UpdateWorkGroupSizeLabels();
            UpdatePixelCohortSizeCombo();
            break;
        case 3:
            OGLDemo->setRenderMode(Application::OpenCL_CPU);
            UpdateWorkGroupSizeLabels();
            UpdatePixelCohortSizeCombo();
            OGLDemo->setWorkgroupSize(this-
void MainComponent::sliderValueChanged (Slider* sliderThatWasMoved)
{
    if (sliderThatWasMoved == TraceDepthSlider)
    {
        OpenGLWindow* OGLDemo;
        assert(OGLDemo = dynamic_cast<OpenGLWindow*> (OGLComponent));
        OGLDemo->setMaxTraceDepth((int)sliderThatWasMoved->getValue());
    }
}

} // [MiscUserCode] You can add your own definitions of your custom methods or any other code here...

void MainComponent::ResetRayIntersectionsNumber() 
{
void MainComponent::ResetRayMissesNumber() {
    this->RayMissNumberLabel2->setText("--", true);
}

void MainComponent::SetRayMissesNumber(int number) {
    juce::String numberString = juce::String(number);
    this->RayMissNumberLabel2->setText(numberString, true);
}

void MainComponent::SetRayIntersectionsNumber(int number) {
    juce::String numberString = juce::String(number);
    this->RayIntersectionNumberLabel2->setText(numberString, true);
}

size_t MainComponent::GetRaytraceKernelWorkgroupSize() {
    return (size_t)OpenCLRelated::OpenCLManager::Instance()-
    >QueryRayTraceKernelRecommendedWorkGroupSize();
}

void MainComponent::UpdatePixelCohortSizeCombo()
{
    uint maxSize;
    uint minSize;
    PixelCohortSizeCombo->clear(true);
    OpenGLWindow* OGLDemo;
    assert(OGLDemo = dynamic_cast<OpenGLWindow*> (OGLComponent));
    // Find the max work group size.
    maxSize = 262144;  // Hard coded for resolution - power of 2
    if (OGLDemo->getRenderMode() == Application::GCC_CPU_Recursive ||
        OGLDemo->getRenderMode() == Application::GCC_CPU_Iterative) {
        minSize = 1;
    } else {
        minSize = this->GetRaytraceKernelWorkgroupSize();
    }
    // Update the combo box
    int itemID = 1;
    for (uint s = maxSize; s >= minSize; s /= 2) {
        PixelCohortSizeCombo->addItem(juce::String(s), itemID++);
    }
    PixelCohortSizeCombo->setSelectedId(1, false);
}

void MainComponent::UpdateWorkGroupSizeLabels() {
    OpenGLWindow* OGLDemo;
    assert(OGLDemo = dynamic_cast<OpenGLWindow*> (OGLComponent));
    // Find the max work group size.
    juce::String sizeString;
    if (OGLDemo->getRenderMode() == Application::GCC_CPU_Recursive ||
        OGLDemo->getRenderMode() == Application::GCC_CPU_Iterative) {
        sizeString = "--";
    } else {
        sizeString = juce::String((int)this->GetRaytraceKernelWorkgroupSize());
    }
    this->WorkGroupSizeLabel2->setText(sizeString, true);
}

//
//==============================================================================
#if 0
/*  -- Jucer information section --

This code contains custom code, such as:

- Functions to set the RayIntersectionNumber and RayMissesNumber
- A method to get the RaytraceKernelWorkgroupSize
- Methods to update the PixelCohortSizeCombo and WorkGroupSizeLabels

These custom functions are used to manipulate the user interface elements, such as labels and combo boxes, to display the number of ray intersections and misses. The code also includes checks for rendering modes and resolution to determine the appropriate workgroup size.

The Jucer information section is a placeholder for additional comments or annotations, which are not visible in this section.

*/
#endif

//[MiscUserCode]

#endif
/*  -- Jucer information section --

This code contains custom code, such as:

- Functions to set the RayIntersectionNumber and RayMissesNumber
- A method to get the RaytraceKernelWorkgroupSize
- Methods to update the PixelCohortSizeCombo and WorkGroupSizeLabels

These custom functions are used to manipulate the user interface elements, such as labels and combo boxes, to display the number of ray intersections and misses. The code also includes checks for rendering modes and resolution to determine the appropriate workgroup size.

The Jucer information section is a placeholder for additional comments or annotations, which are not visible in this section.

*/
#endif
This is where the Jucer puts all of its metadata, so don’t change anything in here!

BEGIN_JUCER_METADATA

<JUCER_COMPONENT documentType="Component" className="MainComponent"
componentName=""
parentClasses="public Component" constructorParams=""
variableInitialisers=""
snapPixels="8" snapActive="1" snapShown="1"
overlayOpacity="0.330000013"
    fixedSize="0" initialWidth="1200" initialHeight="800">
    <BACKGROUND backgroundColour="ffdbdbdb"/>
    <LABEL name="ImplementationLabel" id="762b2a2b543a5bb2"
        memberName="ImplementationLabel"
        virtualName="" explicitFocusOrder="0" pos="16 24 150 24"
        edTextCol="ff000000" edBkgCol="0" labelText="Calculation Method :
        editableSingleClick="0" editableDoubleClick="0" focusDiscardsChanges="0"
        font="
            fontsize="15" bold="0" italic="0" justification="33"/>
    <COMBOBOX name="ImplementationCombo" id="625b1f91da1cf241"
        memberName="ImplementationCombo"
        virtualName="" explicitFocusOrder="0" pos="168 24 192 24"
        editable="0"
        layout="33" items="" textWhenNonSelected="" textWhenNoItems="(no
        choices)"/>
    <GENERICCOMPONENT name="OGLComponent" id="f4fcd9e709dfa7"
        memberName="OGLComponent"
        virtualName="" explicitFocusOrder="0" pos="16 24 150 24"
        params=""/>
</JUCER_COMPONENT>

END_JUCER_METADATA
*/
#endif
*/

This is an automatically generated file created by the Jucer!

Creation date:  15 Dec 2010 9:46:10pm

Be careful when adding custom code to these files, as only the code within the "//[xyz]" and "//[/xyz]" sections will be retained when the file is loaded and re-saved.

Jucer version: 1.12

The Jucer is part of the JUCE library - "Jules' Utility Class Extensions"
Copyright 2004-6 by Raw Material Software ltd.

#ifndef __JUCER_HEADER_MAINCOMPONENT_MAINCOMPONENT_3C8D1CEA__
#define __JUCER_HEADER_MAINCOMPONENT_MAINCOMPONENT_3C8D1CEA__
// [Headers] -- You can add your own extra header files here --
#include "gui_headers.h"
#include "common.h"
#include "OpenCLManager.h"
// [Headers]

//==============================================================================
/**
 // [Comments]
 An auto-generated component, created by the Jucer.

 Describe your class and how it works here!

 // [Comments]
 */

class MainComponent : public Component,
                    public ComboBoxListener,
                    public SliderListener,
                    public ButtonListener
{
public:

//==============================================================================

    MainComponent ();
    ~MainComponent ();

//==============================================================================

// [UserMethods] -- You can add your own custom methods in this section.
// [UserMethods]

    void paint (Graphics& g);
    void resized();
    void comboBoxChanged (ComboBox* comboBoxThatHasChanged);
    void sliderValueChanged(Slider* sliderThatWasMoved);
    void buttonClicked (Button* buttonThatWasClicked);

//==============================================================================

// (prevent copy constructor and operator= being generated..)

// [UserVariables] -- You can add your own custom variables in this section.
// [UserVariables]

Label* ImplementationLabel;
ComboBox* ImplementationCombo;
Label* PixelCohortSizeLabel;
ComboBox* PixelCohortSizeCombo;
Label* WorkGroupSizeLabel;
Label* WorkGroupSizeLabel2;
Label* TraceDepthLabel;
Slider* TraceDepthSlider;
TextButton* RenderButton;
Component* OGLComponent;
Label* RayIntersectionNumberLabel;
Label* RayIntersectionNumberLabel2;
Label* RayMissNumberLabel;
Label* RayMissNumberLabel2;

// (prevent copy constructor and operator= being generated..)
MainComponent (const MainComponent&);
const MainComponent& operator= (const MainComponent&);

void ResetRayIntersectionsNumber();
void ResetRayMissesNumber();
void SetRayIntersectionsNumber(int number);
void SetRayMissesNumber(int number);
size_t GetRaytraceKernelWorkgroupSize();
void UpdatePixelCohortSizeCombo();
void UpdateWorkGroupSizeLabels();

};

#endif   // __JUCER_HEADER_MAINCOMPONENT_MAINCOMPONENT_3C8D1CEA__

// OpenCLManager.cpp
// 2011 - Gary Deng - dengsta@gmail.com
// Manages OpenCL resources

#include "OpenCLManager.h"

namespace OpenCLRelated {

using namespace Application;

// Singleton
OpenCLManager* OpenCLManager::m_Instance = NULL;

// Public
OpenCLManager::OpenCLManager(void) {
    this->InitDevice();
    this->InitContext();
    this->InitCommandQueue();
    this->InitPrograms();
}

OpenCLManager::~OpenCLManager(void) {
    clReleaseCommandQueue(m_CommandQueueCPU);
    clReleaseContext(m_ContextCPU);
    clReleaseCommandQueue(m_CommandQueueGPU);
    clReleaseContext(m_ContextGPU);
    for (std::map<std::string, cl_kernel>::iterator it =
        m_MapKernelNameToKernel.begin();
        it != m_MapKernelNameToKernel.end(); it++)
        { clReleaseKernel(it->second); }
}

OpenCLManager* OpenCLManager::Instance() {
    if (m_Instance)
        return m_Instance;
    m_Instance = new OpenCLManager();
    return m_Instance;
}

int OpenCLManager::device_stats(cl_device_id device_id) {
    int err;
    size_t returned_size;

    // Report the device vendor and device name
    cl_char vendor_name[1024] = {0};
cl_char device_name[1024] = {0};
cl_char device_profile[1024] = {0};
cl_char device_extensions[1024] = {0};
cl_device_local_mem_type local_mem_type;

cl_ulong global_mem_size, global_mem_cache_size;
cl_ulong max_mem_alloc_size;

cl_uint clock_frequency, vector_width, max_compute_units;

size_t max_work_item_dims, max_work_group_size, max_work_item_sizes[3];

cl_uint vector_types[] = {
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR,
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT,
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG,
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE
};

char *vector_type_names[] = {
    "char", "short", "int", "long", "float", "double"
};

err = clGetDeviceInfo(device_id, CL_DEVICE_VENDOR, sizeof(vendor_name),
vendor_name, &returned_size);
err|= clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(device_name),
device_name, &returned_size); 
err|= clGetDeviceInfo(device_id, CL_DEVICE_PROFILE,
sizeof(device_profile), device_profile, &returned_size); 
err|= clGetDeviceInfo(device_id, CL_DEVICE_EXTENSIONS, 
sizeof(device_extensions), device_extensions, &returned_size); 
err|= clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_TYPE,
sizeof(local_mem_type), &local_mem_type, &returned_size);

err|= clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE, 
sizeof(global_mem_size), &global_mem_size, &returned_size);
err|= clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE,
sizeof(global_mem_cache_size), &global_mem_cache_size, &returned_size);
err|= clGetDeviceInfo(device_id, CL_DEVICE_MAX_MEM_ALLOC_SIZE,
sizeof(max_mem_alloc_size), &max_mem_alloc_size, &returned_size);

err|= clGetDeviceInfo(device_id, CL_DEVICE_MAX_CLOCK_FREQUENCY,
sizeof(clock_frequency), &clock_frequency, &returned_size);

err|= clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
sizeof(max_work_group_size), &max_work_group_size, &returned_size);

err|= clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
sizeof(max_work_item_dims), &max_work_item_dims, &returned_size);

err|= clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES,
sizeof(max_work_item_sizes), max_work_item_sizes, &returned_size);

err|= clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS,
sizeof(max_compute_units), &max_compute_units, &returned_size);

printf("Vendor: %s\n", vendor_name);
print("Device Name: %s\n", device_name);
printf("Profile: %s\n", device_profile);
printf("Supported Extensions: %s\n", device_extensions);

printf("Local Mem Type (Local=1, Global=2): %i\n", (int)local_mem_type);
printf("Global Mem Size (MB): %li\n", (int)global_mem_size/(1024*1024));
printf("Global Mem Cache Size (Bytes): %li\n", (int)global_mem_cache_size);
printf("Max Mem Alloc Size (MB): %ld\n", (long int)max_mem_alloc_size/(1024*1024));

printf("Clock Frequency (MHz): %i\n", clock_frequency);

for(int i=0; i<6; i++){
    err|= clGetDeviceInfo(device_id, vector_types[i], 
    sizeof(clock_frequency), &vector_width, &returned_size);
printf("Vector type width for: %s = %i\n", vector_type_names[i], vector_width);
}

printf("\nMax Work Group Size: %lu\n", max_work_group_size);
//printf("Max Work Item Dims: %lu\n", max_work_item_dims);
//for(size_t i=0; i<max_work_item_dims; i++)
// printf("Max Work Items in Dim %lu: %lu\n", (long unsigned) (i+1), (long unsigned) max_work_item_sizes[i]);

printf("Max Compute Units: %i\n", max_compute_units);
printf("\n");
return CL_SUCCESS;
}

void OpenCLManager::InitializePixelDataMem(PixelDataStructOpenCL* pds_ocl) {
    cl_context m_Context = (this->m_DeviceType == OpenCL_CPU) ? m_ContextCPU : m_ContextGPU;
    size_t uintBufferSize;
    uintBufferSize = sizeof(unsigned int) * pds_ocl->size;
    // Set up device buffers.
    pds_ocl->color = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, uintBufferSize, NULL, NULL);
}

void OpenCLManager::InitializePrimitiveDataMem(SphereDataStructCpp* sds_cpp, PlaneDataStructCpp* plds_cpp,
                                               SphereDataStructOpenCL* sds_ocl, PlaneDataStructOpenCL* plds_ocl) {
    cl_context m_Context = (this->m_DeviceType == OpenCL_CPU) ? m_ContextCPU : m_ContextGPU;
    cl_device_id m_Device = (this->m_DeviceType == OpenCL_CPU) ? m_DeviceCPU : m_DeviceGPU;
    cl_int err = 0;
    // Initialize Plane Data
    size_t floatBufferSize = sizeof(float) * pds_cpp->size;
    size_t intBufferSize = sizeof(int) * pds_cpp->size;
    // Set up device buffers.
    plds_ocl->normalX = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->normalX, CL_TRUE, 0, floatBufferSize, plds_cpp->normalX, 0, NULL, NULL);
    plds_ocl->normalY = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->normalY, CL_TRUE, 0, floatBufferSize, plds_cpp->normalY, 0, NULL, NULL);
    plds_ocl->normalZ = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->normalZ, CL_TRUE, 0, floatBufferSize, plds_cpp->normalZ, 0, NULL, NULL);
    plds_ocl->d = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->d, CL_TRUE, 0, floatBufferSize, plds_cpp->d, 0, NULL, NULL);
    plds_ocl->diffuse = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->diffuse, CL_TRUE, 0, floatBufferSize, plds_cpp->diffuse, 0, NULL, NULL);

plds_ocl->specular = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->specular, CL_TRUE, 0, floatBufferSize, plds_cpp->specular, 0, NULL, NULL);

plds_ocl->reflection = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->reflection, CL_TRUE, 0, floatBufferSize, plds_cpp->reflection, 0, NULL, NULL);

plds_ocl->refraction = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->refraction, CL_TRUE, 0, floatBufferSize, plds_cpp->refraction, 0, NULL, NULL);

plds_ocl->refrIndex = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->refrIndex, CL_TRUE, 0, floatBufferSize, plds_cpp->refrIndex, 0, NULL, NULL);

plds_ocl->red = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->red, CL_TRUE, 0, floatBufferSize, plds_cpp->red, 0, NULL, NULL);

plds_ocl->green = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->green, CL_TRUE, 0, floatBufferSize, plds_cpp->green, 0, NULL, NULL);

plds_ocl->blue = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->blue, CL_TRUE, 0, floatBufferSize, plds_cpp->blue, 0, NULL, NULL);

plds_ocl->isLight = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, intBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, plds_ocl->isLight, CL_TRUE, 0, floatBufferSize, plds_cpp->isLight, 0, NULL, NULL);

assert(err == CL_SUCCESS);
// End Plane Data

// Initialize Sphere Data
floatBufferSize = sizeof(float) * sds_cpp->size;
intBufferSize = sizeof(int) * sds_cpp->size;
// Set up device buffers.

sds_ocl->centerX = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err = clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->centerX, CL_TRUE, 0, floatBufferSize, sds_cpp->centerX, 0, NULL, NULL);

sds_ocl->centerY = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->centerY, CL_TRUE, 0, floatBufferSize, sds_cpp->centerY, 0, NULL, NULL);

sds_ocl->centerZ = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->centerZ, CL_TRUE, 0, floatBufferSize, sds_cpp->centerZ, 0, NULL, NULL);

sds_ocl->recRadius = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, floatBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->recRadius, CL_TRUE, 0, floatBufferSize, sds_cpp->recRadius, 0, NULL, NULL);
102

>recRadius, 0, NULL, NULL);
    sds_ocl->sqRadius = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->sqRadius, CL_TRUE,
    0, floatBufferSize, sds_cpp->recRadius, 0, NULL, NULL);

>sqRadius, 0, NULL, NULL);
    sds_ocl->diffuse = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->diffuse, CL_TRUE,
    0, floatBufferSize, sds_cpp->diffuse, 0, NULL, NULL);

>diffuse, 0, NULL, NULL);
    sds_ocl->specular = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->specular, CL_TRUE,
    0, floatBufferSize, sds_cpp->specular, 0, NULL, NULL);

>specular, 0, NULL, NULL);
    sds_ocl->reflection = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->reflection, CL_TRUE,
    0, floatBufferSize, sds_cpp->reflection, 0, NULL, NULL);

>reflection, 0, NULL, NULL);
    sds_ocl->refraction = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->refraction, CL_TRUE,
    0, floatBufferSize, sds_cpp->refraction, 0, NULL, NULL);

>refraction, 0, NULL, NULL);
    sds_ocl->refrIndex = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->refrIndex, CL_TRUE,
    0, floatBufferSize, sds_cpp->refrIndex, 0, NULL, NULL);

>refrIndex, 0, NULL, NULL);
    sds_ocl->red = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->red, CL_TRUE,
    0, floatBufferSize, sds_cpp->red, 0, NULL, NULL);

>red, 0, NULL, NULL);
    sds_ocl->green = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->green, CL_TRUE,
    0, floatBufferSize, sds_cpp->green, 0, NULL, NULL);

>green, 0, NULL, NULL);
    sds_ocl->blue = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    floatBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->blue, CL_TRUE,
    0, floatBufferSize, sds_cpp->blue, 0, NULL, NULL);

>blue, 0, NULL, NULL);
    sds_ocl->isLight = clCreateBuffer(m_Context, CL_MEM_READ_WRITE,
    intBufferSize, NULL, NULL);
    err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->isLight, CL_TRUE,
    0, floatBufferSize, sds_cpp->isLight, 0, NULL, NULL);

>isLight, 0, NULL, NULL);
    assert(err == CL_SUCCESS);
    // End Sphere Data

void OpenCLManager::InitializeStatisticsDataMem(
    StatisticsDataStructCpp* sds_cpp,
    StatisticsDataStructOpenCL* sds_ocl) {

    cl_device_id m_Device = (this->m_DeviceType == OpenCL_CPU) ? m_DeviceCPU :
    m_DeviceGPU;
    cl_context m_Context = (this->m_DeviceType == OpenCL_CPU) ? m_ContextCPU :
    m_ContextGPU;
    cl_command_queue m_CommandQueue = (this->m_DeviceType == OpenCL_CPU) ?
    m_CommandQueueCPU : m_CommandQueueGPU;

    size_t intBufferSize = sizeof(int) * sds_cpp->size;
cl_int err = 0;
// Set up device buffers.
sds_ocl->rayIntersectionsCount = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, intBufferSize, NULL, NULL);
err = clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->rayIntersectionsCount, CL_TRUE, 0, intBufferSize, sds_cpp->rayIntersectionsCount, 0, NULL, NULL);
sds_ocl->rayMissesCount = clCreateBuffer(m_Context, CL_MEM_READ_WRITE, intBufferSize, NULL, NULL);
err |= clEnqueueWriteBuffer(m_CommandQueue, sds_ocl->rayMissesCount, CL_TRUE, 0, intBufferSize, sds_cpp->rayMissesCount, 0, NULL, NULL);
assert(err == CL_SUCCESS);
}

void OpenCLManager::DisposePixelDataMem(PixelDataStructOpenCL* pds_ocl) {
    clReleaseMemObject(pds_ocl->color);
}

void OpenCLManager::DisposePrimitiveDataMem(PlaneDataStructOpenCL* plds_ocl, SphereDataStructOpenCL* sds_ocl) {
    // Dispose of planes
    clReleaseMemObject(plds_ocl->normalX);
    clReleaseMemObject(plds_ocl->normalY);
    clReleaseMemObject(plds_ocl->normalZ);
    clReleaseMemObject(plds_ocl->d);
    clReleaseMemObject(plds_ocl->diffuse);
    clReleaseMemObject(plds_ocl->specular);
    clReleaseMemObject(plds_ocl->reflection);
    clReleaseMemObject(plds_ocl->refraction);
    clReleaseMemObject(plds_ocl->refrIndex);
    clReleaseMemObject(plds_ocl->red);
    clReleaseMemObject(plds_ocl->green);
    clReleaseMemObject(plds_ocl->blue);
    clReleaseMemObject(plds_ocl->isLight);

    // Dispose of spheres
    clReleaseMemObject(sds_ocl->centerX);
    clReleaseMemObject(sds_ocl->centerY);
    clReleaseMemObject(sds_ocl->centerZ);
    clReleaseMemObject(sds_ocl->recRadius);
    clReleaseMemObject(sds_ocl->sqRadius);
    clReleaseMemObject(sds_ocl->diffuse);
    clReleaseMemObject(sds_ocl->specular);
    clReleaseMemObject(sds_ocl->reflection);
    clReleaseMemObject(sds_ocl->refraction);
    clReleaseMemObject(sds_ocl->refrIndex);
    clReleaseMemObject(sds_ocl->red);
    clReleaseMemObject(sds_ocl->green);
    clReleaseMemObject(sds_ocl->blue);
    clReleaseMemObject(sds_ocl->isLight);
}

void OpenCLManager::DisposeStatisticsDataMem(StatisticsDataStructOpenCL* sds_ocl) {
    clReleaseMemObject(sds_ocl->rayIntersectionsCount);
    clReleaseMemObject(sds_ocl->rayMissesCount);
}

void OpenCLManager::RayTraceKernel(
    int maxDepth,
    float camPosX, float camPosY,
    float camPosZ, int framebufferWidth, int
firstPixelIndex,  
float screenWorldBoundX1, float  
screenWorldBoundY1,  
float worldDX, float worldDY,  
PixelDataStructOpenCL* pds_ocl,  
PlaneDataStructOpenCL* plds_ocl,  
SphereDataStructOpenCL* sds_ocl,  
StatisticsDataStructOpenCL* stds_ocl) {
    cl_int err = 0;
    cl_kernel kernel = GetKernelByName(RAYTRACE_KERNEL);
    int arg = 0, paramCount = 0;
    size_t shared_spheres_size, shared_planes_size;
    int pixels_param_count, planes_param_count, spheres_param_count;

    // Set kernel arguments
    err |= clSetKernelArg(kernel, arg++, sizeof(int), &maxDepth);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &camPosX);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &camPosY);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &camPosZ);
    err |= clSetKernelArg(kernel, arg++, sizeof(int), &frameBufferWidth);
    err |= clSetKernelArg(kernel, arg++, sizeof(int), &firstPixelIndex);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &screenWorldBoundX1);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &screenWorldBoundY1);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &worldDX);
    err |= clSetKernelArg(kernel, arg++, sizeof(float), &worldDY);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &pds_ocl->color);
    pixels_param_count = arg - paramCount;
    paramCount = arg;
    err |= clSetKernelArg(kernel, arg++, sizeof(size_t), &sds_ocl->size);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->centerX);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->centerY);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->centerZ);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->recRadius);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->sqRadius);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->diffuse);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->specular);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->reflection);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->refraction);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->refrIndex);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->red);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->green);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->blue);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &sds_ocl->isLight);
    spheres_param_count = arg - paramCount;
    paramCount = arg;
    err |= clSetKernelArg(kernel, arg++, sizeof(size_t), &plds_ocl->size);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->normalX);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->normalY);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->normalZ);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->d);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->diffuse);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->specular);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->reflection);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->refraction);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->refrIndex);
    err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->isLight);
refrIndex);
err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->red);
err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->green);
err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->blue);
err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &plds_ocl->isLight);
planes_param_count = arg - paramCount;
paramCount = arg;
err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &stds_ocl->rayIntersectionsCount);
err |= clSetKernelArg(kernel, arg++, sizeof(cl_mem), &stds_ocl->rayMissesCount);
shared_spheres_size = (spheres_param_count * sds_ocl->size * 4);
shared_planes_size = (planes_param_count * plds_ocl->size * 4);
err |= clSetKernelArg(kernel, arg++, shared_spheres_size, NULL);
err |= clSetKernelArg(kernel, arg++, shared_planes_size, NULL);
assert(err == CL_SUCCESS);

// Inspection BEFORE kernel calculations
// ReportPixelData(pds_ocl);

// Run the calculation by enqueueing it and forcing
// the command queue to complete the task.
size_t global_work_size = pds_ocl->size;
err = clEnqueueNDRangeKernel(m_CommandQueue, kernel, 1, NULL,
&global_work_size, &m_raytraceKernelWorkGroupSize, 0, NULL, NULL);
assert(err == CL_SUCCESS);

// Inspection AFTER kernel calculations
// ReportPixelData(pds_ocl);
}

void OpenCLManager::ReadPixelData(PixelDataStructOpenCL* pds_ocl, unsigned int* destPixelBuffer) {
cl_command_queue m_CommandQueue = (this->m_DeviceType == OpenCL_CPU) ?
m_CommandQueueCPU : m_CommandQueueGPU;

cl_int err = 0;
size_t bufferSize = sizeof(Pixel) * pds_ocl->size;
err |= clEnqueueReadBuffer(m_CommandQueue, pds_ocl->color, CL_TRUE, 0,
bufferSize, destPixelBuffer, 0, NULL, NULL);
assert(err == CL_SUCCESS);
}

void OpenCLManager::ReadStatisticsData(StatisticsDataStructOpenCL* sds_ocl,
StatisticsDataStructCpp* sds_cpp) {
cl_command_queue m_CommandQueue = (this->m_DeviceType == OpenCL_CPU) ?
m_CommandQueueCPU : m_CommandQueueGPU;

cl_int err = 0;
size_t bufferSize = sizeof(int) * sds_ocl->size;
err |= clEnqueueReadBuffer(m_CommandQueue, sds_ocl->rayIntersectionsCount, CL_TRUE, 0,
rayIntersectionsCount, 0, NULL, NULL);
err |= clEnqueueReadBuffer(m_CommandQueue, sds_ocl->rayMissesCount, CL_TRUE, 0,
rayMissesCount, 0, NULL, NULL);
assert(err == CL_SUCCESS);
}
// Testing
void OpenCLManager::ReportPixelData(PixelDataStructOpenCL& pds_ocl) {
    cl_command_queue m_CommandQueue = (this->m_DeviceType == OpenCL_CPU) ? m_CommandQueueCPU : m_CommandQueueGPU;
    if (DEBUG) {
        struct {
            int* lastPrimObjType;
            int* lastPrimID;
            float* rayPosX;
            float* rayPosY;
            float* rayPosZ;
            float* rayDirX;
            float* rayDirY;
            float* rayDirZ;
            float* rayDist;
            float* color;
        } ss;
        size_t size = pds_ocl.size;
        ss.lastPrimObjType = new int[size];
        ss.lastPrimID = new int[size];
        ss.rayPosX = new float[size];
        ss.rayPosY = new float[size];
        ss.rayPosZ = new float[size];
        ss.rayDirX = new float[size];
        ss.rayDirY = new float[size];
        ss.rayDirZ = new float[size];
        ss.rayDist = new float[size];
        ss.color = new float[size];
        cl_int err = 0;
        size_t floatBufferSize, intBufferSize, uintBufferSize;
        floatBufferSize = sizeof(float) * pds_ocl.size;
        intBufferSize = sizeof(int) * pds_ocl.size;
        uintBufferSize = sizeof(unsigned int) * pds_ocl.size;
        // Read results from buffers
        err |= clEnqueueReadBuffer(m_CommandQueue, pds_ocl.color, CL_TRUE, 0,
                                  uintBufferSize,
                                  (void*)ss.color, 0, NULL,
                                  NULL);
        assert(err == CL_SUCCESS);
        clFinish(m_CommandQueue);
        std::cout << "<--GPU Values: PixelData-->
        for (int i = 0; i < pds_ocl.size; i++) {
            std::cout
                << i << "": "
                << "lastPrimObjType" << ss.lastPrimObjType[i] << "" "
                << "lastPrimID" << ss.lastPrimID[i] << "" "
                << "rayPosX" << ss.rayPosX[i] << "" "
                << "rayPosY" << ss.rayPosY[i] << "" "
                << "rayPosZ" << ss.rayPosZ[i] << "" "
                << "rayDirX" << ss.rayDirX[i] << "" "
                << "rayDirY" << ss.rayDirY[i] << "" "
                << "rayDirZ" << ss.rayDirZ[i] << "" "
                << "rayDist" << ss.rayDist[i] << "" "
                << "color" << ss.color[i] << "\n";
        }
    }
}
void OpenCLManager::SetDevice(OpenCLDevice device) {
    // Initialize stuff
this->m_DeviceType = device;

c_int err = 0;
size_t returned_size = 0;
cl_device_id currentDevice = (m_DeviceType == OpenCL_CPU) ? this->m_DeviceCPU :
this->m_DeviceGPU;
// Get some information about the returned device
cl_char vendor_name[1024] = {0};
cl_char device_name[1024] = {0};
err = clGetDeviceInfo(currentDevice, CL_DEVICE_VENDOR,
sizeof(vendor_name),
vendor_name, &returned_size);
err |= clGetDeviceInfo(currentDevice, CL_DEVICE_NAME, sizeof(device_name),
device_name, &returned_size);
assert(err == CL_SUCCESS);
printf("Connecting to %s %s...\n", vendor_name, device_name);
// Get lots of info about the device
device_stats(currentDevice);
}

void OpenCLManager::SetRaytraceKernelWorkGroupSize(size_t a_Size) {
  m_raytraceKernelWorkGroupSize = a_Size;
}

size_t OpenCLManager::QueryRayTraceKernelRecommendedWorkGroupSize() {
  cl_device_id m_Device = (this->m_DeviceType == OpenCL_CPU) ? m_DeviceCPU :
m_DeviceGPU;
  size_t maxSize;
  cl_int err;
  cl_kernel kernel = GetKernelByName(RAYTRACE_KERNEL);
  err = clGetKernelWorkGroupInfo(kernel, m_Device,
CL_KERNEL_WORK_GROUP_SIZE, sizeof(maxSize), &maxSize, NULL);
  if (err != CL_SUCCESS)
  {
    printf("Error: Failed to retrieve kernel work group info! %d\n", err);
    exit(1);
  }
  return maxSize;
}

// Private
cl_kernel OpenCLManager::GetKernelByName(std::string a_kernelName) {
  std::string kernelName(a_kernelName);
  switch (m_DeviceType) {
  case OpenCL_CPU:
    kernelName += " OpenCL_CPU";
    break;
  case OpenCL_GPU:
    kernelName += " OpenCL_GPU";
    break;
  }

  if (m_MapKernelNameToKernel[kernelName] == 0) {
    cl_int err = 0;
    m_MapKernelNameToKernel[kernelName] = clCreateKernel(m_MapKernelNameToProgram[kernelName],
a_kernelName.c_str(), &err);
    assert(err == CL_SUCCESS);
  }

  return m_MapKernelNameToKernel[kernelName];
}

void OpenCLManager::MapKernelNameToProgram(std::string a_kernelName, std::string
a_programName) {
    // Build CPU programs
    {
        cl_int err = 0;
        std::string programName(a_programName);
        std::string kernelName(a_kernelName);
        programName += " OpenCL_CPU";
        kernelName += " OpenCL_CPU";
        cl_program program = m_MapProgramNameToProgram[programName];

        if (!program) {
            // Load the source code to a string.
            char* programSource = this->LoadProgramSource(a_programName.c_str());

            // Create the program with source string.
            program = clCreateProgramWithSource(m_ContextCPU, 1, (const char**)&programSource, NULL, &err);
            assert(err == CL_SUCCESS);

            // Build the program.
            err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
            char build[2048];
            clGetProgramBuildInfo(program, m_DeviceCPU, CL_PROGRAM_BUILD_LOG, 2048, build, NULL);
            printf("Build Log:\n%s\n", build); // Prints any build errors
            assert(err == CL_SUCCESS);

            m_MapProgramNameToProgram[programName] = program;
        }
    }

    m_MapKernelNameToProgram[kernelName] = program;
}

// Build GPU programs
{
    cl_int err = 0;
    std::string programName(a_programName);
    std::string kernelName(a_kernelName);
    programName += " OpenCL_GPU";
    kernelName += " OpenCL_GPU";
    cl_program program = m_MapProgramNameToProgram[programName];

    if (!program) {
        // Build the program, since it has not yet been built.
        // Load the source code to a string.
        char* programSource = this->LoadProgramSource(a_programName.c_str());

        // Create the program with source string.
        program = clCreateProgramWithSource(m_ContextGPU, 1, (const char**)&programSource, NULL, &err);
        assert(err == CL_SUCCESS);

        // Build the program.
        err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
        char build[2048];
        clGetProgramBuildInfo(program, m_DeviceGPU, CL_PROGRAM_BUILD_LOG, 2048, build, NULL);
        printf("Build Log:\n%s\n", build); // Prints any build errors
        assert(err == CL_SUCCESS);

        m_MapProgramNameToProgram[programName] = program;
    }
}
m_MapKernelNameToProgram[kernelName] = program;
}

char* OpenCLManager::LoadProgramSource(const char* filename) {
    struct stat statbuf;
    FILE *fh;
    char *source;

    fh = fopen(filename, "r");
    if (fh == 0)
        return 0;

    stat(filename, &statbuf);
    source = (char *) malloc(statbuf.st_size + 1);
    fread(source, statbuf.st_size, 1, fh);
    source[statbuf.st_size] = '\0';

    return source;
}

void OpenCLManager::InitContext() {
    cl_int err = 0;
    // Create a context to perform our calculation with the
    // specified device
    m_ContextCPU = clCreateContext(0, 1, &m_DeviceCPU, NULL, NULL, &err);
    assert(err == CL_SUCCESS);
    m_ContextGPU = clCreateContext(0, 1, &m_DeviceGPU, NULL, NULL, &err);
    assert(err == CL_SUCCESS);
}

void OpenCLManager::InitCommandQueue() {
    m_CommandQueueCPU = clCreateCommandQueue(m_ContextCPU, m_DeviceCPU, 0,
                                             NULL);
    m_CommandQueueGPU = clCreateCommandQueue(m_ContextGPU, m_DeviceGPU, 0,
                                             NULL);
}

// TODO: give some better way of notifying whether or not this was
// successful.
void OpenCLManager::InitDevice() {
    // Init CPU device
    cl_int err = 0;
    // Find the CPU CL device, as a fallback
    err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &this->m_DeviceCPU,
                          NULL);
    std::cout << err << std::endl;
    assert(err == CL_SUCCESS);

    // Init GPU device
    // Find the GPU CL device, this is what we really want
    // If there is no GPU device is CL capable, fall back to CPU
    err = clGetDeviceIds(NULL, CL_DEVICE_TYPE_GPU, 1, &this->m_DeviceGPU,
                          NULL);
    if (err != CL_SUCCESS) this->m_DeviceGPU = this->m_DeviceCPU;
    assert(m_DeviceGPU);

    // Get some information about the returned device
    std::vector<char> vendor_name(1024); std::vector<char> device_name(1024);
    err = clGetDeviceInfo(m_Device, CL_DEVICE_VENDOR, sizeof(vendor_name),
                          &vendor_name[0], &returned_size);
    assert(err == CL_SUCCESS);
    std::cout << "Vendor Name: " << vendor_name[0]) << std::endl;

    err = clGetDeviceInfo(m_Device, CL_DEVICE_NAME, sizeof(device_name),
                          &device_name[0], &returned_size);
    assert(err == CL_SUCCESS);
    std::cout << "Device Name: " << device_name[0] << std::endl;

    printf("Connecting to %s %s...\n", vendor_name, device_name);
}
void OpenCLManager::InitPrograms( void ) {
    cl_int err = 0;
    // Construct the KernelName -> Program map.
    this->MapKernelNameToProgram(RAYTRACE_KERNEL, RAY_TRACER_PROGRAM);
}

openclmanager.h
2011 - Gary Deng - dengsta@gmail.com
#ifndef I_APPLICATION_OPENCLMANAGER_H
#define I_APPLICATION_OPENCLMANAGER_H
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <OpenCL/OpenCL.h>
#include <sys/stat.h>
#include <assert.h>
#include "app_common.h"

namespace OpenCLRelated {

// CL program source defines
#define RAY TRACER_PROGRAM      "raytracer.cl"
#define RAYTRACE KERNEL       "raytrace"

enum OpenCLDevice{
    OpenCL_CPU = 0,
    OpenCL_GPU = 1
};

class OpenCLManager {
public:
    static OpenCLManager* Instance(); // Instantiates and loads OpenCL programs.
    ~OpenCLManager( void);
    void InitializePixelDataMem(Application::PixelDataStructOpenCL* pds_ocl);
    void InitializePrimitiveDataMem(Application::SphereDataStructCpp* sds_CPP,
                                           Application::PlaneDataStructCpp* plds_CPP,
                                           Application::SphereDataStructOpenCL* sds_ocl,
                                           Application::PlaneDataStructOpenCL* plds_ocl);
    void DisposePixelDataMem(Application::PixelDataStructOpenCL* pds_ocl);
    void DisposePrimitiveDataMem(Application::SphereDataStructOpenCL* sds_ocl,
                                           Application::PlaneDataStructOpenCL* plds_ocl);
    void InitializeStatisticsDataMem(Application::StatisticsDataStructCpp* sds_CPP,
                                           Application::StatisticsDataStructOpenCL* sds_ocl);
    void DisposeStatisticsDataMem(Application::StatisticsDataStructOpenCL* sds_ocl);}
void RayTraceKernel(int maxDepth,
  float camPosX, float camPosY,
  float camPosZ,
  int frameBufferWidth, int
  firstPixelIndex,
  float screenWorldBoundX1, float
  screenWorldBoundY1,
  float worldDX, float worldDY,

  Application::PixelDataStructOpenCL* pds_ocl,
  Application::PlaneDataStructOpenCL* plds_ocl,
  Application::SphereDataStructOpenCL* sds_ocl);

// Read data from OpenCL memories to Cpp host memories.
void ReadPixelData(Application::PixelDataStructOpenCL* pds_ocl, unsigned
  int* destPixelBuffer);

// Find the max local size for the ray trace kernel.
size_t QueryRayTraceKernelRecommendedWorkGroupSize();

private:
  OpenCLManager(void); // Private so it can not be called.
  int device_stats(cl_device_id device_id);
  void MapKernelNameToProgram(std::string kernelName, std::string
    programName);
  void InitContext(void);
  void InitCommandQueue(void);
  void InitDevice();
  void InitPrograms(void);
  char* LoadProgramSource(const char* fileName);
  cl_kernel GetKernelByName(std::string kernelName);

static OpenCLManager* m_Instance;
std::map<std::string, cl_program> m_MapProgramNameToProgram;
std::map<std::string, cl_program> m_MapKernelNameToProgram;
std::map<std::string, cl_kernel> m_MapKernelNameToKernel;
std::vector<cl_program> m_Programs;
cl_context m_ContextCPU;
cl_context m_ContextGPU;
cl_command_queue m_CommandQueueCPU;
cl_command_queue m_CommandQueueGPU;
OpenCLDevice m_DeviceType;
cl_device_id m_DeviceGPU;
cl_device_id m_DeviceCPU;
size_t m_raytraceKernelWorkGroupSize;

// Testing
void ReportPixelData(Application::PixelDataStructOpenCL& pds_ocl);

*/

This file is part of the JUCE library - "Jules' Utility Class Extensions"
Copyright 2004-9 by Raw Material Software Ltd.
JUCE can be redistributed and/or modified under the terms of the GNU General Public License (Version 2), as published by the Free Software Foundation. A copy of the license is included in the JUCE distribution, or can be found online at www.gnu.org/licenses.

JUCE is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

---

To release a closed-source product which uses JUCE, commercial licenses are available: visit www.rawmaterialsoftware.com/juce for more information.

---

#include "OpenGLWindow.h"

using namespace Application;
using namespace Common;

Application::Surface* surface = 0;
Pixel* buffer = 0;
Application::Engine* tracer = 0;

//==============================================================================
//class DemoOpenGLCanvas  : public OpenGLComponent//,
//                          public Timer
//{
//public:
//    DemoOpenGLCanvas::DemoOpenGLCanvas()
//    {
//        // (On the iPhone, choose a format without a depth buffer)
//        setPixelFormat (OpenGLPixelFormat (8, 8, 0, 0));
//        startTimer (0);
//        renderMode = Application::GCC_CPU_Recursive;
//    }
//}

DemoOpenGLCanvas::DemoOpenGLCanvas()
{
    // when the component creates a new internal context, this is called, and
    // we'll use the opportunity to create the textures needed.
    void DemoOpenGLCanvas::newOpenGLContextCreated()
    {
        // (no need to call makeCurrentContextActive(), as that will have
        // been done for us before the method call).
        //
        // OpenGL calls.
        clearColor(0.0, 0.0, 0.0, 0.0);
        disable(GL_DEPTH_TEST);
        viewport(0, 0, (GLsizei)getWidth(), (GLsizei)getHeight()); // look into
        this next!
    }
}
// Flip the draw buffer
glRasterPos2i(-1.0, 1.0);    // Set position to bottom left of window.
glPixelZoom(1.0f, -1.0f);

// NonOpenGL calls.
// prepare output canvas
surface = new Application::Surface( OGL_SCRWIDTH, OGL_SCRHEIGHT );
buffer = surface->GetBuffer();
surface->Clear( 0 );
surface->InitCharset();
surface->Print( "timings:", 2, 2, 0xffffffff );
// prepare renderer
tracer = new Application::Engine();
tracer->GetScene()->InitScene();
tracer->SetTarget( surface->GetBuffer(), OGL_SCRWIDTH, OGL_SCRHEIGHT );
}

void DemoOpenGLCanvas::mouseDrag (const MouseEvent& e) {
}

void DemoOpenGLCanvas::resized() {
}

void DemoOpenGLCanvas::renderOpenGL() {
#if ! JUCE_IPHONE
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawPixels( OGL_SCRWIDTH, OGL_SCRHEIGHT, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
    glutSwapBuffers();
    // Check for errors.
    if (glGetError())
        std::cout << "GL error status: " << glGetError() << std::endl;
#endif
}

// GUI interaction functions..
Application::RenderModes DemoOpenGLCanvas::getRenderMode() {
    return renderMode;
}

int DemoOpenGLCanvas::getRayIntersectionsCount() {
    return tracer->GetRayIntersectionsCount();
}

int DemoOpenGLCanvas::getRayMissesCount() {
    return tracer->GetRayMissesCount();
}

void DemoOpenGLCanvas::setRenderMode(Application::RenderModes a_mode) {
    renderMode = a_mode;
    // Initialize relevant devices.
    switch (renderMode) {
    case Application::GCC_CPU_Recursive:
        this->renderMode = Application::GCC_CPU_Recursive;
        break;
    case Application::GCC_CPU_Iterative:
        this->renderMode = Application::GCC_CPU_Iterative;
        break;
    }
case Application::OpenCL_CPU:
  OpenCLRelated::OpenCLManager::Instance()->SetDevice(OpenCLRelated::OpenCL_CPU);
  break;

case Application::OpenCL_GPU:
  OpenCLRelated::OpenCLManager::Instance()->SetDevice(OpenCLRelated::OpenCL_GPU);
  break;
}

void DemoOpenGLCanvas::setMaxTraceDepth(int a_MaxDepth) {
  tracer->SetMaxTraceDepth(a_MaxDepth);
}

void DemoOpenGLCanvas::setPixelBatchSize(int a_PixelBatchSize) {
  tracer->SetCohortSize(a_PixelBatchSize);
}

void DemoOpenGLCanvas::setWorkgroupSize(int a_WorkgroupSize) {
  tracer->SetWorkgroupSize(a_WorkgroupSize);
}

void DemoOpenGLCanvas::render() {
  // Clear the buffer.
  surface->Clear(0);
  int tpos = 0;
  surface->Print( "timings:", 2, 2, 0xffffffff );
  // Do the heavy lifting.
  bool finished;
  tpos = 60;
  // go
  tracer->InitRender();
  tracer->PreRender(renderMode);
  int fstart = Util::Time::GetTimeMilliseconds(); // start timing
  while (!tracer->Render(renderMode))
    renderAndSwapBuffers();
  int ftime = Util::Time::GetTimeMilliseconds() - fstart; // end timing
  tracer->PostRender(renderMode);
  char t[] = "00:00.000";
  t[6] = (ftime / 100) % 10 + '0';
  t[7] = (ftime / 10) % 10 + '0';
  t[8] = (ftime % 10) + '0';
  int secs = (ftime / 1000) % 60;
  int mins = (ftime / 60000) % 100;
  t[3] = ((secs / 10) % 10) + '0';
  t[4] = (secs % 10) + '0';
  t[1] = (mins % 10) + '0';
  t[0] = ((mins / 10) % 10) + '0';
  surface->Print( t, tpos, 2, 0xffffffff );
  tpos += 100;
  renderAndSwapBuffers();
}

OpenGLWindow::OpenGLWindow() {
  setName (T("OpenGL"));
  canvas = new DemoOpenGLCanvas();
  addAndMakeVisible (canvas);
}

OpenGLWindow::~OpenGLWindow() {
  deleteAllChildren();
}
void OpenGLWindow::resized()
{
    canvas->setBounds(10, 10, OGL_SCRWIDTH, OGL_SCRHEIGHT);
}

Application::RenderModes OpenGLWindow::getRenderMode()
{
    return canvas->getRenderMode();
}

int OpenGLWindow::getRayIntersectionsCount()
{
    return canvas->getRayIntersectionsCount();
}

int OpenGLWindow::getRayMissesCount()
{
    return canvas->getRayMissesCount();
}

void OpenGLWindow::setRenderMode(Application::RenderModes mode)
{
    canvas->setRenderMode(mode);
}

void OpenGLWindow::setMaxTraceDepth(int a_MaxDepth)
{
    canvas->setMaxTraceDepth(a_MaxDepth);
}

void OpenGLWindow::setPixelBatchSize(int a_PixelBatchSize)
{
    canvas->setPixelBatchSize(a_PixelBatchSize);
}

void OpenGLWindow::setWorkgroupSize(int a_WorkgroupSize)
{
    canvas->setWorkgroupSize(a_WorkgroupSize);
}

void OpenGLWindow::render()
{
    canvas->render();
}

//}

Component* createOpenGLWindow()
{
    return new OpenGLWindow();
}

#if defined_WIN32
#include <windows.h>
#endif
#include "gui_headers.h"
#include "app_common.h"
#include "surface.h"
#include "ApplicationMain.h"
#include "scene.h"
#include "Util.h"
#include "OpenCLManager.h"

#if JUCE_OPENGL
#if JUCE_WINDOWS
#include <gl/gl.h>
#include <gl/glu.h>
#elif JUCE_LINUX
#include <GL/gl.h>
#include <GL/glut.h>
#else JUCE_IPHONE
#include <OpenGLES/ES1/gl.h>
#include <OpenGLES/ES1/glext.h>
#elif JUCE_MAC
#include <GLUT/glut.h>
#else JUCE_IPHONE
#endif
#endif

#ifndef GL_BGRA_EXT
#define GL_BGRA_EXT 0x80e1
#endif

//==============================================================================
class DemoOpenGLCanvas  : public OpenGLComponent
{

public:
  DemoOpenGLCanvas();
  ~DemoOpenGLCanvas();
  // when the component creates a new internal context, this is called, and
  // we'll use the opportunity to create the textures needed.
  void newOpenGLContextCreated();
  void mouseDrag (const MouseEvent& e);
  void resized();
  void renderOpenGL();
  // GUI interaction functions..
  Application::RenderModes getRenderMode();
  int getRayIntersectionsCount();
  int getRayMissesCount();
  void setRenderMode(Application::RenderModes a_mode);
  void setMaxTraceDepth(int a_MaxDepth);
  void setPixelBatchSize(int a_WorkgroupSize);
  void setWorkgroupSize(int a_WorkgroupSize);
  void render();

private:
  Application::RenderModes renderMode;
};

//==============================================================================
class OpenGLWindow  : public Component
{

};
void setRenderMode(Application::RenderModes mode);
void setMaxTraceDepth(int a_MaxDepth);
void setPixelBatchSize(int a_PixelBatchSize);
void setWorkgroupSize(int a_WorkgroupSize);
void render();
}

//==============================================================================
#endif
// -----------------------------------------------------------
// scene.cpp
// 2011 - modified - Gary Deng - dengsta@gmail.com
// 2004 - original - Jacco Bikker - jacco@bik5.com - www.bik5.com - <>
// Scene classes and data
// -----------------------------------------------------------
#include "common.h"
#include "string.h"
#include "scene.h"
#include "ApplicationMain.h"
#include <vector>
#include <iostream>
namespace Application {

// -----------------------------------------------------------
// Primitive class implementation
// -----------------------------------------------------------
void Primitive::SetName( char* a_Name )
{
    delete m_Name;
    m_Name = new char[strlen( a_Name ) + 1];
    strcpy( m_Name, a_Name );
}

// -----------------------------------------------------------
// Material class implementation
// -----------------------------------------------------------
Material::Material() :
    m_Color( Color( 0.2f, 0.2f, 0.2f ) ),
    m_Ref1( 0 ), m_Diff( 0.2f ), m_Spec( 0.8f ), m_RIndex( 1.5f )
{
}

// Sphere primitive methods
// -----------------------------------------------------------
int Sphere::Intersect( Ray& a_Ray, float& a_Dist )
{
    vector3 v = a_Ray.GetOrigin() - m_Centre;
    float b = -DOT( v, a_Ray.GetDirection() );
    float det = (b * b) - DOT( v, v ) + m_SqRadius;
    int retval = MISS;
    if (det > 0)
    {
        det = sqrtf( det );
        float i1 = b - det;
        float i2 = b + det;
        if (i2 > 0)
        {
if (i1 < 0)
{
    if (i2 < a_Dist)
    {
        a_Dist = i2;
        retval = INPRIM;
    }
    else
    {
        if (i1 < a_Dist)
        {
            a_Dist = i1;
            retval = HIT;
        }
    }
}
return retval;

// --------------------------------------
// Plane primitive class implementation
// --------------------------------------

int PlanePrim::Intersect(Ray& a_Ray, float& a_Dist)
{
    float d = DOT(m_Plane.N, a_Ray.GetDirection());
    if (d != 0)
    {
        float dist = -(DOT(m_Plane.N, a_Ray.GetOrigin()) + m_Plane.D) / d;
        if (dist > 0)
        {
            if (dist < a_Dist)
            {
                a_Dist = dist;
                return HIT;
            }
        }
    }
    return MISS;
}

vector3 PlanePrim::GetNormal(vector3& a_Pos)
{
    return m_Plane.N;
}

// --------------------------------------
// Scene class implementation
// --------------------------------------

Scene::~Scene()
{
    delete m_Primitive;
}

void Scene::InitScene()
{
    m_Primitive = new Primitive*[500];
    m_Sphere = new Sphere*[500];
    m_Plane = new PlanePrim*[500];
    int prim = 0;
    // ground plane
    vector3 TempVector3_1(0.0f, 1.0f, 0.0f);
    m_Primitive[prim] = new PlanePrim(TempVector3_1, 4.4f);
    m_Primitive[prim]->SetName("plane");
    m_Primitive[prim]->GetMaterial()->SetReflection(0.15f);
m_Primitive[prim]->GetMaterial()->SetRefraction( 0.0f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 1.0f );
Color TempColor_1(0.4f, 0.3f, 0.3f);
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_1 );
m_Plane[m_Planes++] = (PlanePrim*)m_Primitive[0]);
prim++;

// big sphere
vector3 TempVector3_2(2.0f, 0.8f, 3.0f);
m_Primitive[prim] = new Sphere( TempVector3_2, 2.5f );
m_Primitive[prim]->SetName( "big sphere" );
m_Primitive[prim]->GetMaterial()->SetReflection( 0.2f );
m_Primitive[prim]->GetMaterial()->SetRefraction( 0.8f );
m_Primitive[prim]->GetMaterial()->SetRefrIndex( 1.3f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 0.1f );
Color TempColor_2(0.7f, 0.7f, 1.0f);
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_2 );
prim++;

m_Sphere[m_Spheres++] = (Sphere*)m_Primitive[1]);

// medium sphere
vector3 TempVector3_3( -5.5f, -0.5, 7.0f );
m_Primitive[prim] = new Sphere( TempVector3_3, 2.0f );
m_Primitive[prim]->SetName( "medium sphere" );
m_Primitive[prim]->GetMaterial()->SetReflection( 0.5f );
m_Primitive[prim]->GetMaterial()->SetRefraction( 0.0f );
m_Primitive[prim]->GetMaterial()->SetRefrIndex( 1.3f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 0.1f );
Color TempColor_3( 0.7f, 0.7f, 1.0f);
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_3 );
prim++;

// extra sphere
vector3 TempVector3_6( -1.5f, -3.8f, 1.0f );
m_Primitive[prim] = new Sphere( TempVector3_6, 1.5f );
m_Primitive[prim]->SetName( "extra sphere" );
m_Primitive[prim]->GetMaterial()->SetReflection( 0.1f );
m_Primitive[prim]->GetMaterial()->SetRefrIndex( 1.3f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 0.25f );
Color TempColor_6( 1.0f, 0.4f, 0.4f);
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_6 );
prim++;  
// back plane
vector3 TempVector3_7( 0.4f, 0.0f, -1.0f );
m_Primitive[prim] = new PlanePrim( TempVector3_7, 12.0f );
m_Primitive[prim]->SetName( "back plane" );
m_Primitive[prim]->GetMaterial()->SetReflection( 0.025f );
m_Primitive[prim]->GetMaterial()->SetRefraction( 0.0f );
m_Primitive[prim]->GetMaterial()->SetSpecular( 0.0f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 0.6f );
Color TempColor_7( 0.5f, 0.3f, 0.5f );
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_7 );
m_Plane[m_Planes++] = ( ( PlanePrim*)m_Primitive[prim]);
prim++;  
// ceiling plane
vector3 TempVector3_8(0.0f, -1.0f, 0.0f);
m_Primitive[prim] = new PlanePrim( TempVector3_8, 7.4f );
m_Primitive[prim]->SetName( "ceiling plane" );
m_Primitive[prim]->GetMaterial()->SetReflection( 0.0f );
m_Primitive[prim]->GetMaterial()->SetRefraction( 0.0f );
m_Primitive[prim]->GetMaterial()->SetSpecular( 0.0f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 0.5f );
Color TempColor_8( 0.4f, 0.7f, 0.7f );
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_8 );
m_Plane[m_Planes++] = ( ( PlanePrim*)m_Primitive[prim]);
prim++;  
// back grid
for ( int x = 0; x < 8; x++ ) for ( int y = 0; y < 7; y++ )
{
    vector3 TempVector3_9( -4.5f + x * 1.5f, -4.3f + y * 1.5f, 10.0f );
m_Primitive[prim] = new Sphere( TempVector3_9, 0.3f );
m_Primitive[prim]->SetName( "back grid sphere" );
m_Primitive[prim]->GetMaterial()->SetReflection( 0.0f );
m_Primitive[prim]->GetMaterial()->SetRefraction( 0.0f );
m_Primitive[prim]->GetMaterial()->SetSpecular( 0.6f );
m_Primitive[prim]->GetMaterial()->SetDiffuse( 0.6f );
Color TempColor_7( 0.3f, 1.0f, 0.4f );
m_Primitive[prim]->GetMaterial()->SetColor( TempColor_7 );
m_Sphere[m_Spheres++] = ( ( Sphere*)m_Primitive[prim]);
prim++;  
// set number of primitives
m_Primitives = prim;

  // For debugging purposes
  std::cout << "<-- SCENE PRIMITIVES DATA --> \n";
  for (int i = 0; i < 500; i++)
  {
    PlanePrim* p = m_Plane[i];
    if (p == NULL) break;
    
    std::cout << "PLANE: " << p->GetNormal().x << " 
";
    <normalX=" << p->GetNormal().x << " 
";
    "normalY=" << p->GetNormal().y << " 
";
    "normalZ=" << p->GetNormal().z << " 
";
    "d=" << p->GetD() << " 
";
    "diffuse=" << p->GetMaterial()->GetDiffuse() << " 
";
    "specular=" << p->GetMaterial()->GetSpecular() << " 
";
    "reflection=" << p->GetMaterial()->GetReflection() << " 
";
    "refraction=" << p->GetMaterial()->GetRefraction() << " 
";
    "red=" << p->GetMaterial()->GetColor().r << " 
";
    "green=" << p->GetMaterial()->GetColor().g << " 
";
    "blue=" << p->GetMaterial()->GetColor().b << " 
";
    "isLight=" << p->IsLight() << "\n";
  }
  for (int i = 0; i < 500; i++)
  {
    Sphere* s = m_Sphere[i];
    if (s == NULL) break;
    
    std::cout << "SPHERE: " << s->GetNormal().x << " 
";
    <normalX=" << s->GetNormal().x << " 
";
    "normalY=" << s->GetNormal().y << " 
";
    "normalZ=" << s->GetNormal().z << " 
";
    "radius=" << s->GetRadius() << " 
";
    "diffuse=" << s->GetMaterial()->GetDiffuse() << " 
";
    "specular=" << s->GetMaterial()->GetSpecular() << " 
";
    "reflection=" << s->GetMaterial()->GetReflection() << " 
";
    "refraction=" << s->GetMaterial()->GetRefraction() << " 
";
    "red=" << s->GetMaterial()->GetColor().r << " 
";
    "green=" << s->GetMaterial()->GetColor().g << " 
";
    "blue=" << s->GetMaterial()->GetColor().b << " 
";
    "isLight=" << s->IsLight() << "\n";
  }
}
std::cout << "SPHERE: " << s->GetCentre().x << " " << s->GetCentre().y << " " << s->GetCentre().z << " " << s->GetNormal().x << " " << s->GetNormal().y << " " << s->GetNormal().z << " " << s->GetSqRadius() << " " << s->GetMaterial()->GetDiffuse() << " " << s->GetMaterial()->GetSpecular() << " " << s->GetMaterial()->GetReflection() << " " << s->GetMaterial()->GetRefraction() << " " << s->GetMaterial()->GetColor().r << " " << s->GetMaterial()->GetColor().g << " " << s->GetMaterial()->GetColor().b << " " << "isLight=" << s->IsLight() << "\n";
}
}; // namespace Application

#include "app_common.h"
#include "ApplicationMain.h"
#include "string.h"
#include <vector>
namespace Application {
    // Intersection method return values
    #define HIT   1  // Ray hit primitive
    #define MISS  0  // Ray missed primitive
    #define INPRIM -1  // Ray started inside primitive

    // Material class definition
    class Material {
    public:
    Material();
    void SetColor( Color& a_Color ) { m_Color = a_Color; }
    Color GetColor() { return m_Color; }
    void SetDiffuse( float a_Diff ) { m_Diff = a_Diff; }
    void SetSpecular( float a_Spec ) { m_Spec = a_Spec; }
    void SetReflection( float a_Refl ) { m_Refl = a_Refl; }
    void SetRefraction( float a_Refr ) { m_Refr = a_Refr; }
    float GetSpecular() { return m_Spec; }
    float GetDiffuse() { return m_Diff; }
    float GetReflection() { return m_Refl; }
    float GetRefraction() { return m_Refr; }
    void SetRefrIndex( float a_Refr ) { m_RIndex = a_Refr; }
    float GetRefrIndex() { return m_RIndex; }
    private:
}
Color m_Color;
float m_Refl, m_Refr;
float m_Diff, m_Spec;
float m_RIndex;
};

// Primitive class definition

class Primitive
{
public:
enum
{
SPHERE = 1,
PLANE
};
Primitive() : m_Name( 0 ), m_Light( false ) {}
Material* GetMaterial() { return &m_Material; }
void SetMaterial( Material& a_Mat ) { m_Material = a_Mat; }
virtual int GetType() = 0;
virtual int Intersect( Ray& a_Ray, float& a_Dist ) = 0;
// virtual Color GetColor( vector3& ) { return m_Material.GetColor(); }
virtual void Light( bool a_Light ) { m_Light = a_Light; }
bool IsLight() { return m_Light; }
voidSetName( char* a_Name );
char* GetName() { return m_Name; }
protected:
Material m_Material;
char* m_Name;
bool m_Light;
};

// Sphere primitive class definition

class Sphere : public Primitive
{
public:
int GetType() { return SPHERE; }
Sphere( vector3& a_Centre, float a_Radius ) :
m_Centre( a_Centre ), m_SqRadius( a_Radius * a_Radius ),
m_Radius( a_Radius ), m_RRadius( 1.0f / a_Radius ) {};
vector3& GetCentre() { return m_Centre; }
float GetSqRadius() { return m_SqRadius; }
float GetRecRadius() { return m_RRadius; }
int Intersect( Ray& a_Ray, float& a_Dist )
{
return (a_Pos - m_Centre) * m_RRadius;
}
private:
vector3 m_Centre;
float m_SqRadius, m_Radius, m_RRadius;
};

// PlanePrim primitive class definition

class PlanePrim : public Primitive
{
public:
int GetType() { return PLANE; }
PlanePrim( vector3& a_Normal, float a_D ) : m_Plane( plane( a_Normal, a_D ) ) {};
}
vector3& GetNormal() { return m_Plane.N; }
float GetD() { return m_Plane.D; }
int Intersect( Ray& a_Ray, float& a_Dist );
vector3 GetNormal( vector3& a_Pos );

private:
  plane m_Plane;
};

// Scene class definition
class Scene
{
public:
  Scene() : m_Primitives( 0 ), m_Primitive( 0 ), m_Spheres( 0 ), m_Planes ( 0 );
  ~Scene();
  void SphereTree( int& a_Prim, float a_Radius, vector3 a_Pos, int a_Depth );
  void InitScene();
  int GetNrPrimitives() { return m_Primitives; }
  Primitive* GetPrimitive( int a_Idx ) { return m_Primitive[a_Idx]; }
private:
  int m_Primitives, m_Spheres, m_Planes;
  Primitive** m_Primitive;
  Sphere** m_Sphere;
  PlanePrim** m_Plane;
};

// namespace Application

// surface.cpp
#include "common.h"
#include "surface.h"
#include "stdio.h"
#include "string.h"
namespace Application {

Surface::Surface( int a_Width, int a_Height ) :
  m_Width( a_Width ),
  m_Height( a_Height )
{
  m_Buffer = new Pixel[a_Width * a_Height];
}
Surface::~Surface()
{
  delete [] m_Buffer;
}
void Surface::Clear( Pixel a_Color )
{
  int s = m_Width * m_Height;
  for ( int i = 0; i < s; i++ ) m_Buffer[i] = a_Color;
}
void Surface::Print( char* a_String, int x1, int y1, Pixel color )
{
    Pixel* t = m_Buffer + x1 + y1 * m_Width;
    int i;
    for ( i = 0; i < (int)(strlen( a_String )); i++ )
    {
        long pos = 0;
        if ((a_String[i] >= 'A') && (a_String[i] <= 'Z')) pos =
            s_Transl[a_String[i] - ('A' - 'a')];
        else pos = s_Transl[a_String[i]];
        Pixel* a = t;
        char* c = (char*)s_Font[pos];
        int h, v;
        for ( v = 0; v < 5; v++ )
        {
            for ( h = 0; h < 5; h++ )
            if (*c++ == 'o') *(a + h) = color;
            a += m_Width;
        }
        t += 6;
    }
}

void Surface::SetChar( int c, char* c1, char* c2, char* c3, char* c4, char* c5 )
{
    strcpy( s_Font[c][0], c1 );
    strcpy( s_Font[c][1], c2 );
    strcpy( s_Font[c][2], c3 );
    strcpy( s_Font[c][3], c4 );
    strcpy( s_Font[c][4], c5 );
}

void Surface::InitCharset()
{
    SetChar( 0, "::ooo::", "::ooo::", ::ooo::", ::ooo::", ::ooo::" );
    SetChar( 1, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 2, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 3, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 4, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 5, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 6, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 7, "0000::", "0000::", "0000::", "0000::", "0000::" );
    SetChar( 8, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar( 9, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(10, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(11, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(12, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(13, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(14, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(15, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(16, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(17, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(18, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(19, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(20, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(21, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(22, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(23, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(24, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(25, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(26, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(27, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(28, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(29, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(30, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(31, ::000::, ::000::, ::000::, ::000::, ::000:: );
    SetChar(32, ::000::, ::000::, ::000::, ::000::, ::000:: );
SetChar(33, "ooooo", ":::o:", ":::o:", ":::o:", ":::o:" );
SetChar(34, "::oo:", ":::o:", "::oo:", "::oo:", "::oo:" );
SetChar(35, "::oo:", ":::o:", "::oo:", "::oo:", "::oo:" );
SetChar(36, "::oo:", ":::o:", "::oo:", "::oo:", "::oo:" );
SetChar(37, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(38, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(39, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(40, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(41, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(42, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(43, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(44, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(45, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(46, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(47, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(48, "::oo:", ":::o:", "::oo:", "::oo:" );
SetChar(49, "::oo:", ":::o:", "::oo:", "::oo:" );

# define _APPLICATION_SURFACE_H
#include "string.h"
#include "app_common.h"

namespace Application {

class Surface {

    enum {
        OWNER = 1
    };

public:
    // constructor / destructors
    Surface( int a_Width, int a_Height );
    Surface( char* a_File );
    ~Surface();

    // member data access
    Pixel* GetBuffer() { return m_Buffer; }
    int GetWidth() { return m_Width; }
    int GetHeight() { return m_Height; }

    // Special operations
    void InitCharset();
    void SetChar( int c, char* c1, char* c2, char* c3, char* c4, char* c5 );
    void Print( char* a_String, int x1, int y1, Pixel color );
    void Clear( Pixel a_Color );

private:
    // Attributes
    Pixel* m_Buffer;
    int m_Width, m_Height;
// Static attributes for the buildin font
char s_Font[51][5][5];
int s_Transl[256];
}; // namespace Application
#endif
// -----------------------------------------------------------

// Util.cpp
// 2011 - Gary Deng - dengsta@gmail.com
// Various Utilities
// -----------------------------------------------------------

#include "Util.h"

namespace Util {

unsigned long Time::GetTimeMilliseconds() {
  struct timeval tv;
  struct timezone tz;
  struct tm *tm;

  gettimeofday(&tv, &tz);
  tm=localtime(&tv.tv_sec);

  unsigned long ms = 0;
  ms += tm->tm_hour * 60 * 60 * 1000;
  ms += tm->tm_min * 60 * 1000;
  ms += tm->tm_sec * 1000;
  ms += tv.tv_usec / 1000;

  return ms;
}

// StringComparator class
StringComparator* StringComparator::m_instance = NULL;
StringComparator* StringComparator::Instance() {
  if (m_instance) return m_instance;
  m_instance = new StringComparator();
  return m_instance;
}
}; // namespace Util

// -----------------------------------------------------------
// Util.h
// 2011 - Gary Deng - dengsta@gmail.com
// Various Utilities
// -----------------------------------------------------------

#ifndef I_ROOT_UTIL_H
#define I_ROOT_UTIL_H

#include <stdlib.h>
#include <sys/time.h>
#include <string>

namespace Util {

class Time {
 public:
  static unsigned long GetTimeMilliseconds();
};

class StringComparator {
 public:

StringComparator() {}
static StringComparator* Instance();
void ConcatToFirst(std::string s) { m_firstString += s; }
void ConcatToSecond(std::string s) { m_secondString += s; }
void ResetFirst() { m_firstString.clear(); }
void ResetSecond() { m_secondString.clear(); }
int Compare() { return m_firstString.compare(m_secondString); }

private:
std::string m_firstString;
std::string m_secondString;
static StringComparator* m_instance;

StringComparator() : m_firstString(""), m_secondString("") {}
APPENDIX B

Source Code in OpenCL C

// raytracer.cl
// 2011 - Gary Deng - dengsta@gmail.com
// OpenCL Ray-tracer Implementation kernel and related functions

//@ 0.60.0
//@
//@ Defines
//@
//@ Intersection descriptions/types
#define NO_HIT 0
#define PLANE_HIT 1
#define SPHERE_HIT_OUTSIDE 2
#define SPHERE_HIT_INSIDE 3
//@ 0.61.0
//@ Sphere offset multipliers
#define SI_CENTERX 0
#define SI_CENTERY 1
#define SI_CENTERZ 2
#define SI_RECRADIUS 3
#define SI_SQRADIUS 4
#define SI_DIFFUSE 5
#define SI_SPECULAR 6
#define SI.Reflection 7
#define SI_REFRACTION 8
#define SI_REFRINDEX 9
#define SI_RED 10
#define SI_GREEN 11
#define SI_BLUE 12
#define SI_ISLIGHT 13
//@ 0.62.0
//@ Plane offset multipliers
#define PI_NORMALX 0
#define PI_NORMALY 1
#define PI_NORMALZ 2
#define PI_D 3
#define PI_DIFFUSE 4
#define PI_SPECULAR 5
#define PI.Reflection 6
#define PI_REFRACTION 7
#define PI_REFRINDEX 8
#define PI_RED 9
#define PI_GREEN 10
#define PI_BLUE 11
#define PI_ISLIGHT 12
//@ 0.63.0
//@ Other stuff
#define EPSILON 0.001f

//@ 0.64.0
//@ Functions ( Internal )
//@
// float3 add(float3 A, float3 B) {
//     float3 r = (float3) (A.x+B.x, A.y+B.y, A.z+B.z);
//     return r;
// }
//@
// float3 sub(float3 A, float3 B) {
//     float3 r = (float3) (A.x-B.x, A.y-B.y, A.z-B.z);
//     return r;
// }
//@
// float len(float3 A) {
//     float r = sqrt(A.x*A.x+A.y*A.y+A.z*A.z);
//     return r;
// }

float3 mult(float3 A, float scalar)
{
    float3 r = (float3)(A.x*scalar, A.y*scalar, A.z*scalar);
    return r;
}

float3 norm(float3 A)
{
    float l=1/sqrt(A.x*A.x+A.y*A.y+A.z*A.z);
    float3 r = mult(A, l);
    return r;
}

// ToDo: eventually want to pass in radius to sphere structs instead of sqradius
float3 sphere_normal(float3 a_PI, float3 a_Center, float a_ReciprocalRadius)
{
    float3 r = mult( sub(a_PI, a_Center), a_ReciprocalRadius);
    return r;
}

typedef struct {
    float red;
    float green;
    float blue;
} Color;

typedef struct {
    float3 origin;
    float3 direction;
    float hitDistance;
} Ray;

typedef struct {
    int type;
    int index;
} Primitive;

__kernel void raytrace(
    // ITERATIVE DATA
    __constant int maxDepth,
    // WORLD-SCREEN DATA
    __constant float camPosX, __constant float camPosY, __constant float camPosZ,
    __constant int framebufferWidth, __constant int firstPixelIndex,
    __constant float screenWorldBoundXl, __constant float screenWorldBoundY1,
    __constant float worldDX, __constant float worldDY,
    // PIXEL_DATA
    __global unsigned int* pixColor,
    // SPHERE DATA
    int sphereCount,
    // Geometric data
    __global float* s_CenterX, __global float* s_CenterY, __global float* s_CenterZ,
    __global float* s_RecRadius, __global float* s_SqRadius,
    // Material data
    __global float* s_Diffuse, __global float* s_Specular, __global float*
s_Refraction,
    __global float* s_Refraction, __global float* s_RefrIndex,
// Color data
    __global float* s_Red, __global float* s_Green, __global float* s_Blue,
// Light?
    __global int* s_IsLight,

// PLANE DATA
int planeCount,
// Geometric data
    __global float* p_NormalX, __global float* p_NormalY, __global float* p_NormalZ,
    __global float* p_D,
// Material data
    __global float* p_Diffuse, __global float* p_Specular, __global float* p_Reflection,
    __global float* p_Refraction, __global float* p_RefrIndex,
// Color data
    __global float* p_Red, __global float* p_Green, __global float* p_Blue,
// Light?
    __global int* p_IsLight,

// STATISTICS DATA
    __global int* rayIntersectionsCount, __global int* rayMissesCount,

// SHARED MEMORY
    __local float* shared_spheres, __local float* shared_planes

}
// Load planes data to shared memory
for (int iplane = 0; iplane < planeCount; iplane += lsize) {
    if ( (iplane+lsize) > planeCount)
        lsize = planeCount - iplane;

    if ((iplane + lid) < planeCount) {
        shared_planes[iplane+lid + PI_NORMALX*planeCount]  =
        p_NormalX[iplane+lid];
        shared_planes[iplane+lid + PI_NORMALY*planeCount]  =
        p_NormalY[iplane+lid];
        shared_planes[iplane+lid + PI_NORMALZ*planeCount]  =
        p_NormalZ[iplane+lid];
        shared_planes[iplane+lid + PI_D*planeCount]  =
        p_D[iplane+lid];
        shared_planes[iplane+lid + PI_DIFFUSE*planeCount]  =
        p_Diffuse[iplane+lid];
        shared_planes[iplane+lid + PI_SPECULAR*planeCount]  =
        p_Specular[iplane+lid];
        shared_planes[iplane+lid + PI_REFLECTION*planeCount] =
        p_Reflection[iplane+lid];
        shared_planes[iplane+lid + PI_REFRACTION*planeCount] =
        p_Refraction[iplane+lid];
        shared_planes[iplane+lid + PI_REFINDEX*planeCount] =
        p_RefrIndex[iplane+lid];
        shared_planes[iplane+lid + PI_RED*planeCount]   =
        p_Red[iplane+lid];
        shared_planes[iplane+lid + PI_GREEN*planeCount]   =
        p_Green[iplane+lid];
        shared_planes[iplane+lid + PI_BLUE*planeCount]   =
        p_Blue[iplane+lid];
        shared_planes[iplane+lid + PI_ISLIGHT*planeCount]  =
        p_IsLight[iplane+lid];
    }
}

barrier(CLK_LOCAL_MEM_FENCE); // wait until all data are loaded to shared memory.

// USEFUL STUFF
// RayCast stuff
float3 camPos3 = (float3)(camPosX, camPosY, camPosZ);
float3 screenCoord3 = (float3)(
    (firstPixelIndex + gid) % frameBufferWidth * worldDX + screenWorldBoundX1,
    (firstPixelIndex + gid) / frameBufferWidth * worldDY + screenWorldBoundY1,
    0);
Ray rayCast;
rayCast.origin = camPos3;
rayCast.direction = norm(sub(screenCoord3, camPos3));
// Last-primitive-intersected data stuff
Primitive lastPrim;
float3 pi3;
// Color components stuff
Color color = { 0.0f, 0.0f, 0.0f};
// Reflection coefficients
float reflectionCoeffs[17]; // Hard coded to maxDepth = 16
reflectionCoeffs[0] = 1.0f;
// Misc stuff
bool done = false;
int currentDepth = 1;
// For each level of the ray trace tree
while ( (currentDepth <= maxDepth) && (!done) ) {
    // Create temporary color
    Color localColor = { 0.0f, 0.0f, 0.0f };
    // Reset some values
    rayCast.hitDistance = 1000000.0f;
lastPrim.type = NO_HIT;
lastPrim.index = -1;

// ---- Find the nearest intersection ----
for (int p = 0; p < planeCount; p++)
{
    // Intersect with planes
    float3 n3 = (float3)(shared_planes[p + PI_NORMALX*planeCount],
                      shared_planes[p + PI_NORMALY*planeCount],
                      shared_planes[p + PI_NORMALZ*planeCount]);
    float d = dot(n3, rayCast.direction);
    if (d != 0)
    {
        float dist = -(dot(n3, rayCast.origin) + shared_planes[p +
                      PI_D*planeCount]) / d;
        if (dist > 0)
        {  
            if (dist < rayCast.hitDistance)
            {
                rayCast.hitDistance = dist;
                lastPrim.type       = PLANE_HIT;
                lastPrim.index      = p;
            }
        }
    }
}

for (int s = 0; s < sphereCount; s++)
{
    // Intersect with spheres
    float3 c3 = (float3)(shared_spheres[s + SI_CENTERX*sphereCount],
                         shared_spheres[s + SI_CENTERY*sphereCount],
                         shared_spheres[s + SI_CENTERZ*sphereCount]);
    float3 v3 = sub( rayCast.origin, c3);
    float b = -dot(v3, rayCast.direction);
    float det = (b * b) - dot(v3, v3) + shared_spheres[s +
                         SI_SQRADIUS*sphereCount];
    if (det > 0)
    {
        det = sqrt(det);
        float i1 = b - det;
        float i2 = b + det;
        if (i2 > 0)
        {  
            if (i1 < 0)
            {  
                if (i2 < rayCast.hitDistance)
                {  
                    rayCast.hitDistance = i2;
                    lastPrim.type       = SPHERE_HIT_INSIDE;
                    lastPrim.index      = s;
                }
            }
            else
            {  
                if (i1 < rayCast.hitDistance)
                {  
                    rayCast.hitDistance = i1;
                    lastPrim.type       = SPHERE_HIT_OUTSIDE;
                    lastPrim.index      = s;
                }
            }
        }
    }
}

// ---- Handle intersection ----
if (lastPrim.type != NO_HIT)
{  
    // Update statistics
    rayIntersectionsCount[gid]++;

    if ( (lastPrim.type == SPHERE_HIT_OUTSIDE || lastPrim.type == SPHERE_HIT_INSIDE ) &&
        (shared_spheres[lastPrim.index + SI_ISLIGHT*sphereCount] == 1) ) {  
        // Have hit a light, stop tracing
        localColor.red = 1.0f; localColor.green = 1.0f;
        done = true;
    }  
    else {  
        // determine color at point of intersection
        pi3 = add(rayCast.origin, mult(rayCast.direction, rayCast.hitDistance) );

        // trace lights
        for (int l = 0; l < sphereCount; l++)
        {
            if (shared_spheres[l + SI_ISLIGHT*sphereCount] == 1)
            {  
                // If it is a light
                
                // handle point light source
                float3 lc3 = (float3)(shared_spheres[l + SI_CENTERX*sphereCount],
                shared_spheres[l + SI_CENTERY*sphereCount],
                shared_spheres[l + SI_CENTERZ*sphereCount]);

                float shade = 1.0f;
                float3 L3 = sub(lc3, pi3);
                float tdist = len(L3);
                L3 = norm(L3);

                // Create light ray (shadow feeler)
                Ray lightRay;
                lightRay.origin = add(pi3, mult(L3, EPSILON));
                lightRay.direction = L3;

                if (shade > 0)
                {
                    // Check for hit against planes
                    for (int p = 0; p < planeCount; p++)
                    {
                        float3 n3 =
                        (float3)(shared_planes[p + PI_NORMALX*planeCount],
                        shared_planes[p + PI_NORMALY*planeCount],
                        shared_planes[p + PI_NORMALZ*planeCount]);

                        float d = dot(n3, lightRay.direction);
                        if (d != 0)
                        {
                            float dist = -(dot(n3, lightRay.origin) + shared_planes[p + PI_D*planeCount]) / d;
                            if (dist > 0)
                            {
                                if (dist < tdist)
                                {
                                    tdist = dist;
                                    shade = dist;
                                    break;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
if (shade > 0) // If there has not already been a plane hit
{
    // Check for hit against spheres
    for (int s = 0; s < sphereCount; s++)
    {
        if (shared_spheres[s + SI_ISLIGHT*sphereCount] != 0)
            continue; // "if (pr != light)"

        float3 sc3 = (float3)(shared_spheres[s + SI_CENTERX*sphereCount],
        shared_spheres[s + SI_CENTERY*sphereCount],
        shared_spheres[s + SI_CENTERZ*sphereCount]);
        float3 v3 = sub(lightRay.origin,
        lightRay.direction);
        float b = -dot(v3,
        v3) + shared_spheres[s + SI_SQRADIUS*sphereCount];
        float det = (b * b) - dot(v3,
        v3); if (det > 0)
        {
            det = sqrt(det);
            float i1 = b - det;
            float i2 = b + det;
            if (i2 > 0)
            {
                if (i1 < 0)
                {
                    if (i2 < tdist)
                    {
                        tdist = i2;
                        shade = 0;
                        break;
                    }
                    else
                    {
                        if (i1 < tdist)
                        {
                            tdist = i1;
                            shade = 0;
                            break;
                        }
                    }
                }
            }
        }
    }
}

// Shading contribution from light.
if (shade > 0)
{
    float3 lc3 = (float3)(shared_spheres[l + SI_CENTERX*sphereCount],
    shared_spheres[l + SI_CENTERY*sphereCount],
    shared_spheres[l + SI_CENTERZ*sphereCount]);
SI_CENTERZ*sphereCount));
    float3 L3 = sub(lc3, pi3);
    L3 = norm(L3);
    float3 n3;
    // If last prim was a plane...
    if (lastPrim.type == PLANE_HIT) {
        n3 = (float3)(shared_planes[lastPrim.index + PI_NORMALX*planeCount],
                   shared_planes[lastPrim.index + PI_NORMALY*planeCount],
                   shared_planes[lastPrim.index + PI_NORMALZ*planeCount]);
        // calculate DIFFUSE SHADING
        if (shared_planes[lastPrim.index + PI_DIFFUSE*planeCount] > 0) {
            float dot = dot (L3, n3);
            if (dot > 0) {
                float diff = dot *
                shared_planes[lastPrim.index + PI_DIFFUSE*planeCount] * shade;
                // add diffuse component to ray color
                Color ncol = { diff
                              * shared_planes[lastPrim.index + PI_RED*planeCount],
                              diff * shared_planes[lastPrim.index + PI_GREEN*planeCount],
                              diff * shared_planes[lastPrim.index + PI_BLUE*planeCount] };
                // Scale back the color, if necessary
                if (ncol.red > 1.0f || ncol.green > 1.0f || ncol.blue > 1.0f) {
                    float max = 1.0f;
                    if (ncol.red > max) max = ncol.red;
                    if (ncol.green > max) max = ncol.green;
                    if (ncol.blue > max) max = ncol.blue;
                    ncol.red /= max;
                    ncol.green /= max;
                    ncol.blue /= max;
                    ncol.red;
                    ncol.green;
                    ncol.blue;
                }
            }
        }
        // determine SPECULAR COMPONENT
        if (shared_planes[lastPrim.index + PI_SPECULAR*planeCount] > 0) {
            // point light source:
            float3 v3 = rayCast.direction;
            float3 R3 = sub(L3, mult(n3, 2.0f*dot(L3,n3) ) );
            float DOT = dot( v3, R3 );
            if (DOT > 0) {
                float spec = powr(DOT, 20) * shared_planes[lastPrim.index + PI_SPECULAR*planeCount] * shade;
component to ray color
* shared_spheres[1 + SI_RED*sphereCount],
shared_spheres[1 + SI_GREEN*sphereCount],
shared_spheres[1 + SI_BLUE*sphereCount] );
component to be within 0 <-> 1.0
ncol.red = 1.0f; else if (ncol.red < 0.0f) ncol.red = 0.0f;
if (ncol.green > 1.0f) ncol.green = 1.0f; else if (ncol.green < 0.0f) ncol.green = 0.0f;
if (ncol.blue > 1.0f) ncol.blue = 1.0f; else if (ncol.blue < 0.0f) ncol.blue = 0.0f;
localColor.red += ncol.red;
localColor.green += ncol.green;
localColor.blue += ncol.blue;
}

|| lastPrim.type == SPHERE_HIT_OUTSIDE) {
float recRadius =
shared_spheres[lastPrim.index + SI_RECRADIUS*sphereCount];
float3 sphereCenter3 =
(shared_spheres[lastPrim.index + SI_CENTERX*sphereCount],
shared_spheres[lastPrim.index + SI_CENTERY*sphereCount],
shared_spheres[lastPrim.index + SI_CENTERZ*sphereCount]);
n3 = sphere_normal(pi3,
sphereCenter3, recRadius);
// calculate DIFFUSE SHADING
if (shared_spheres[lastPrim.index + SI_DIFFUSE*sphereCount] > 0) {
float DOT = dot (L3, n3);
if (DOT > 0) {
float diff = DOT *
shared_spheres[lastPrim.index + SI_DIFFUSE*sphereCount] * shade;
// add diffuse
component to ray color
* shared_spheres[lastPrim.index + SI_RED*sphereCount],
shared_spheres[lastPrim.index + SI_GREEN*sphereCount],
shared_spheres[lastPrim.index + SI_BLUE*sphereCount] );

|| ncol.green > 1.0f || ncol.blue > 1.0f)
1.0f;
> max) max = ncol.red;
(ncol.green > max) max = ncol.green;
> max) max = ncol.blue;
1.0f/max;
ncol.green *= 1.0f / max;
ncol.blue *= 1.0f / max;

localColor.red += ncol.red;
localColor.green += ncol.green;
localColor.blue += ncol.blue;

if (shared_spheres[lastPrim.index + SI_SPECULAR*sphereCount] > 0) {
    sample once for specular highlight
    rayCast.direction = 
    n3, 2.0f*dot(L3, n3); };

    float3 r3 = sub( L3, mult(n3, 2.0f*dot(L3, n3) ) );
    float DOT = dot( v3, r3 );
    if (DOT > 0) {
        float spec =
        powr(DOT, 20) * shared_spheres[lastPrim.index + SI_SPECULAR*sphereCount] * shade;

        Color ncol = { spec *
                       shared_spheres[l + SI_RED*sphereCount],
                       spec *
                       shared_spheres[l + SI_GREEN*sphereCount],
                       spec *
                       shared_spheres[l + SI_BLUE*sphereCount] };

        // Adjust specular component to be within 0 <-> 1.0
        ncol.red = 1.0f; else if (ncol.red < 0.0f) ncol.red = 0.0f;
        ncol.green = 1.0f; else if (ncol.green < 0.0f) ncol.green = 0.0f;
        ncol.blue = 1.0f; else if (ncol.blue < 0.0f) ncol.blue = 0.0f;

        localColor.red +=
        localColor.green +=
        localColor.blue +=

    }

    } // determine SPECULAR COMPONENT

    // calculate reflection
    float refl;
    if (lastPrim.type == SPHERE_HIT_INSIDE || lastPrim.type ==
        SPHERE_HIT_OUTSIDE) {
        refl = shared_spheres[lastPrim.index +
        SI_REFLECTION*sphereCount];
    }
    if (lastPrim.type == PLANE_HIT) {
        refl = shared_planes[lastPrim.index +
        PI_REFLECTION*planeCount];
    }
if (refl > 0.0f) {
    float3 n3;
    if (lastPrim.type == SPHERE_HIT_INSIDE ||
        lastPrim.type == SPHERE_HIT_OUTSIDE) {
        float recRadius =
            shared_spheres[lastPrim.index + SI_RECRADIUS*sphereCount];
        float3 sphereCenter3 =
            (float3)(shared_spheres[lastPrim.index + SI_CENTERX*sphereCount],
                     shared_spheres[lastPrim.index +
                     SI_CENTERY*sphereCount],
                     shared_spheres[lastPrim.index +
                     SI_CENTERZ*sphereCount]);
        n3 = sphere_normal(pi3, sphereCenter3, recRadius);
        if (lastPrim.type == PLANE_HIT) {
            n3 = (float3)(shared_planes[lastPrim.index +
                           PI_NORMALX*planeCount],
                          shared_planes[lastPrim.index +
                          PI_NORMALY*planeCount],
                          shared_planes[lastPrim.index +
                          PI_NORMALZ*planeCount]);
        }
        float3 r3 = sub(rayCast.direction, mult(dot(
                     rayCast.direction, n3), 2.0f)*n3);
        rayCast.origin = add( mult(r3, EPSILON), pi3);
        rayCast.direction = r3;
        reflectionCoeffs[currentDepth] = refl;
    } else {
        // -- No more reflections needed --
        done = true;
    }
} else {
    // Update statistics
    rayMissesCount[gid]++;
    // -- No intersection occurred; stop tracing --
    done = true;
}

// ---- Adjust contribution of reflected ray ----
// Scale back the local color contribution, if necessary
if (localColor.red > 1.0f || localColor.green > 1.0f ||
    localColor.blue > 1.0f) {
    float max = 1.0f;
    if (localColor.red > max) max = localColor.red;
    if (localColor.green > max) max = localColor.green;
    if (localColor.blue > max) max = localColor.blue;
    localColor.red *= 1.0f/max;
    localColor.green *= 1.0f/max;
    localColor.blue *= 1.0f/max;
}

    // -- Weight the reflection coefficient --
    float rc = reflectionCoeffs[0];
    for (int i = 1; i < currentDepth; i++) { rc *= reflectionCoeffs[i]; }
    localColor.red *= rc;
    localColor.green *= rc;
    localColor.blue *= rc;

    // update color
    color.red += localColor.red;
color.green += localColor.green;
color.blue += localColor.blue;
++currentDepth;
} // end while loop

// Process color values
int r, g, b;
r = (int)(color.red * 256);
g = (int)(color.green * 256);
b = (int)(color.blue * 256);

if (r > 255) r = 255;
if (g > 255) g = 255;
if (b > 255) b = 255;

pixColor[ gid ] = (b << 16) + (g << 8) + r;
APPENDIX C

Permission from Jacco Bikker

Requesting permission to use your ray tracer code for masters' project.

3 messages
Gary Deng <dengsta@gmail.com>  Fri, Feb 19, 2010 at 2:47 PM
To: bikker.j@nhtv.nl

Hello,
I have been looking over your tutorial and source code for your ray tracer on DevMaster.net and think that it is pretty good. I am at the early stages of a masters' project in which I will be writing an educational tool on the topic of ray tracing performance using traditional CPU calculations versus ray tracing performance using GPGPU procedures. I would like to rewrite portions of your ray tracer code using OpenCL calls.
I was wondering if I could use your ray tracer source code as a foundation. If you are OK with this, do you know what stipulations I would need to follow in order to properly do so? It will be a purely educational project. I am not familiar with the licenses that apply to your code at all and any insight would be appreciated.
Thanks!

Jacco Bikker <Bikker.J@nhtv.nl>  Wed, Feb 24, 2010 at 12:06 PM
To: dengsta@gmail.com
Gary ,
Sorry for my late reply.
About the ray tracer: you can definitely use it, without any restrictions, as long as you are not making money with it. :) Are you going to use my Arauna engine, or one of the ray tracers that came with the articles? You can find info about Arauna here: http://igad.nhtv.nl/~bikker . Arauna is a full-fletched real-time ray tracer, used for several projects at our university.
- Jacco.

>>> Gary Deng <dengsta@gmail.com> 02/19/10 11:47 PM >>>
| [Quoted text hidden] |
| ---------------------------------------------------------- Op deze e-mail zijn de volgende voorwaarden van toepassing : The following disclaimer applies to the e-mail message :
| http://www.nhtv.nl/meerweten/disclaimer.htm | ---------------------------------------------------------- |

Gary Deng <dengsta@gmail.com>
To: Jacco Bikker <Bikker.J@nhtv.nl>
Hello Jacco,

Wed, Feb 24, 2010 at 5:22 PM
That's great to hear. I will not be making money on it and it will be strictly for educational purposes. Currently, I am leaning toward using a ray tracers that came with the articles, since they are well-documented (they come with tutorials) and are pretty bare bones (not a lot of extra fluff). However, I will definitely take a look at the Arauna engine to see if I can work with that. I will keep you informed how things go.

Thanks!

Gary Deng <dengsta@gmail.com>
Gary
Gary
[Quoted text hidden]
REFERENCES


