Notes on the Multi-Threaded Skeleton Ray Tracer

Download this multi-threaded version of the ray tracer from Github:

https://github.com/danoli3/Multithreaded-Ray-Tracer

Based on the Textbook "Ray Tracing from the Ground Up" by Kevin Suffern

http://www.raytracegroundup.com/

The original skeleton ray tracer by Sverre Kvaale can be found on the downloads page of the textbook's website:

http://www.raytracegroundup.com/downloads.html

By Daniel Rosser

2/2/2012

Table of Contents

Multi-Threading

The application has been changed to support multi-threading by changing the way the program manages its render queue, and adding in multiple worker threads.

The original skeleton ray tracer as described in Kevin Suffern's book Ray Tracing from the Ground up, renders each image by rendering a single pixel at a time in a loop in a camera class (Pinhole for example) which iterates through each two-dimensional X and Y pixel.

In the multi-threaded version this was changed to support 'jobs'. Jobs are essentially lists of X and Y pixels to render. This change allowed the render loop to segment off different parts of the image to render separately. To manage the different pixels, a Queue system was developed so that the program will will ask for the next job in the list upon completion of a job, or at the start of a render.

The threading system uses wxThreads, a part of the wxWidgets API. Each of the threads, which are created on render start by either being selected automatically by a hardware query on your system (to determine how many cores are available), or manually through the interface, will begin to render a list of jobs provided to it by the Queue system. These threads will be spread across your CPU cores and will be run in parallel. They will therefore render the image a lot quicker than with a single core.

Reference Counting with Smart Pointers

The multi-threaded skeleton program also uses smart pointers and reference counting for managing pointers to prevent memory leaks.

This insures that any object pointer that is created and then stored as a `SmartPointer` will not cause a memory leak when it is deleted.

Mersenne Twister Random Numbers

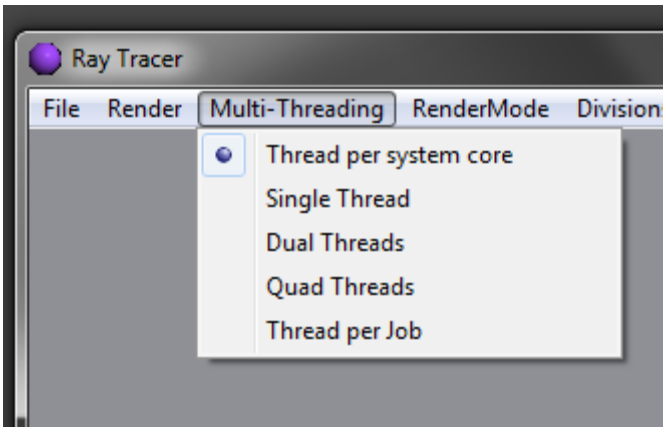
For random numbers, the skeleton program now uses a *pseudorandom number generator* based on Mersenne twisters, which provides fast generation of very high-quality pseudorandom numbers.

The `MTRand` class provides a system independent random number generator that is portable across multiple operating systems, allowing for a random number seed to generate the same result on different computers and operating systems. Optionally, this can be easily changed to the system rand function through a Boolean value passed to the object, or set in the `MTRand` class.

Rendering Options

There have been significant changes to the interface of the multi-threaded skeleton as described below.

Multi-threading Options

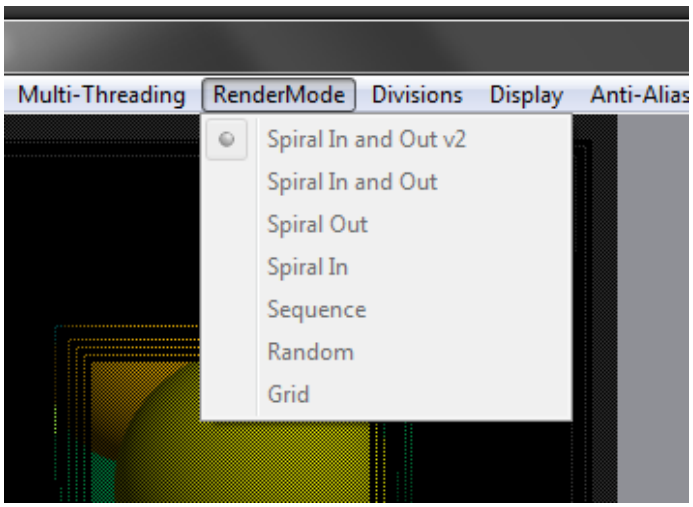


Under the Multi-Threading interface options you can choose how many threads you want to use for rendering. By default, one thread per system core will be selected. The program will find out how many cores you have on your system, including hyper-threaded cores, and use the maximum number of threads. For example, Intel i5 and i7 chips have four hyper-threaded cores, which allow two threads per core, for a total of 8 threads.

The other options are fairly self-explanatory: Single Thread will only use one thread, Dual Threads will use two, Quad Threads will use four.

Thread per Job is an experimental function which will create a thread for each job. This will cause a lot of timesharing on your CPU and potentially a lot of deadlocks. It's not the optimal way to render, but it can produce some nice visual effects.

RenderMode Options



Under the RenderMode menu options you can select the way in which the program will split up the jobs. This will produce many different results rendering, but the final image will always be the same.

Spiral In and Out v2 is the default, as it renders inwards and outwards at the same time but also skips every second pixel, so you get a visually complete image at 50% render. This can be very helpful to see what the image will look like. It will then fill in the final 50% to complete the render.

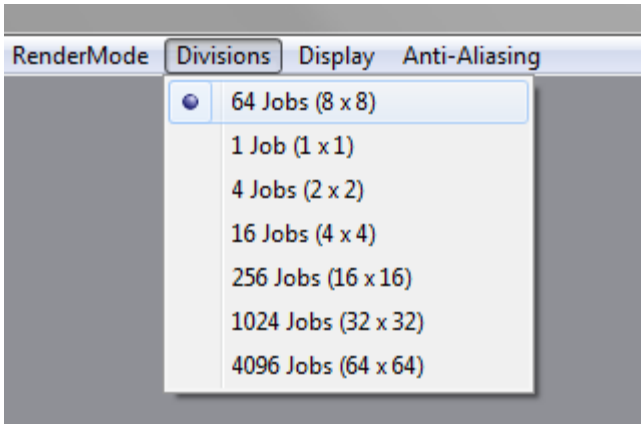
Spiral In and Out will do a full render of outside and in (the same as v2 but it doesn't skip any pixels on the first half). Spiral Out will render the image from the centre outwards. Spiral In will render it from the outside in.

Sequence will set up the jobs in X (across) then Y (up) and will split them by divisions according to the Divisions menu.

Random will render a pixel at random, defined by shuffling a list of pixels like the Sequence and then assigning jobs to an amount of random pixels.

Grid will set up a block of pixels X and Y to render in a square like fashion for each job.

Divisions Options



Under the Divisions menu you can choose how many jobs there will be by how the pixels are divided up. An 8 x 8 division is the default and will give a fairly even break up of pixels with 64 jobs.

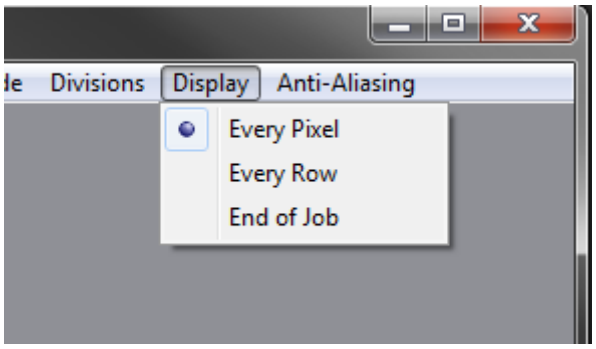
Using 1 x 1 will only create 1 Job and will only be useful if you are using a single thread and do not want any other jobs. It is still recommended to use the 64 Jobs (8 x 8) for a single thread unless you want to simulate the original skeleton program.

A 2 x 2 will split the jobs into 4, which may be good to see the whole screen render at once in a Grid or Sequence rending mode on Quad Threads.

4x4 uses 16 jobs and is recommended if you don't want to use the 8 x 8.

64 x 64 is an experimental setting which will create 4096 jobs in total. This mode gives very interesting visual rending results however it is definitely not recommended for use all the time.

Display Options

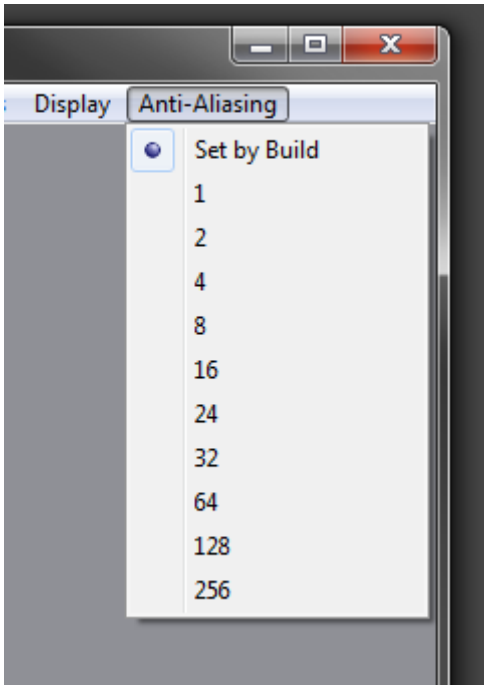


Under the Display menu you can select how often you want the rendering threads to send their pixels back to the interface. This can be used to limit deadlocks between the interface thread accepting pixels, which can be helpful on very fast renders which may lock up the interface on fast computers.

The default setting is Every Pixel like the original skeleton program. Setting this to Every Row will make the job only send-off pixels every 10% of its job. Setting to End of Job will only send the pixels once the entire job that thread had been rendering has been completed.

Keep in mind the Job / Division size when you modify this interface.

Anti-Aliasing Options

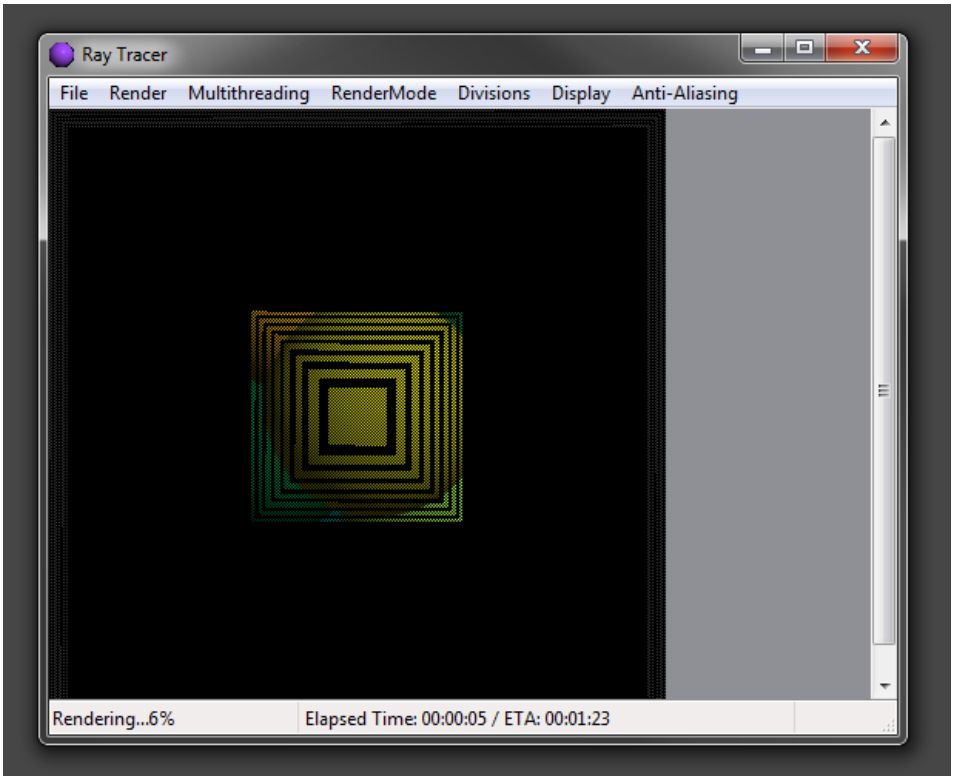


Under the Anti-Aliasing menu you can set and override the antialiasing set by the build function. I added this menu because I often found myself rebuilding and compiling just to change the sample size for the anti-aliasing.

By default this is set to Set by Build which will use the value you have set in your build function. Once the render has started though, the program will find out what that sample size is from the build function and if it is the same as one listed, it will show you by selecting it in the list above.

Setting the sample size to another sample before starting a render will override the build function's sample size.

Other Interface Changes



The rendering progress will be shown in the bottom left of the window. Also, information about the stage in the render will also be shown here, such as building world, or cleaning up.

The elapsed time and estimated time will be shown in the middle tab at the bottom.

As in the original skeleton program the rendering window will be fully re-sizeable, updateable, and scrollable during the rendering process. I have tested this with images up to 12,000 x 12,000 pixels.

File Output Options

The new skeleton program will also allow you to save the image in the following image file types:

.png, .tiff, .jpeg, .bmp

The JPEG will be saved by default with a quality of 100. But if you want to preserve the full image quality, you should save it as a .png or .tiff as these are uncompressed.

Project Files

wxRaytracer.NET2003.sln - Microsoft Visual Studio .NET 2003

wxRaytracerVCPP2005EE.sln - Microsoft Visual C++ 2005 Express Edition (Also works with Professional, Ultimate)

wxRaytracerVCPP2008EE.sln - Microsoft Visual C++ 2008 Express Edition (Also works with Professional, Ultimate)

wxRaytracerVCPP2010EE.sln - Microsoft Visual C++ 2010 Express Edition (Also works with Professional, Ultimate)

Recommendations:

- Use Visual Studio 2010 for the project as I have optimised the Solution file for faster speeds in Release mode.
- When rendering an image that is going to take a long time, always use Release mode unless you are debugging a problem, because there is a speed increase of 2-4 times in Release.
- Start your project with this Skeleton as the base if you want multi-threading, else as stated below, and you can merge them later if you want to.

Other Enhancements in the Multi-threaded Skeleton:

- Multi-threaded Compiling in Visual Studio 2010 Solution.
- Pausing/Resuming the render works
- Queue system to split up screen into Job's
- Closing the application in the middle of a render will not cause memory leaks or crash.
- Enhanced wxWidgets settings

How to Extend a Project Built with the Original Skeleton?

1. Download and extract the multi-threaded Skeleton.
2. Use the multi-threaded Skeleton as the main project, and start adding your custom made classes and files from the original skeleton to the project directory/solution.
3. You will need to Merge and Update the `Pinhole` camera class using the example function `Pinhole::render_scene(const World& w, const std::vector<Pixel>& pixels)`
4. Make sure if you are using any compound objects to read the Notes On Implementation below:

Notes on Implementation

If you implement the geometric ojects `Compound`, `RegularGrid`, or `TriangleMesh`, you need to be careful with the following statement:

```
material_ptr = objects[j]->get_material();
```

Materials cannot be returned the way described in the textbook for these objects using a temporary local variable bound to the upper abstract base class `GeometicObject`, as this object is being used by multiple threads.

In the multi-threaded ray tracer you need to you store the returned material inside the `ShadeRec` object with the code

```
sr.material_ptr = objects[j]->get_material();
```

Source Code Changes from the Original Skeleton

(+ means added new class, - means existing class that has been modified)

User Interface/

-wxraytracer.h/.cpp (Interface changes, Queue System added, multi-threading added, fixed crashes, Pause / End now working)

World/

-World.h/.cpp (Changes for Multi-threading)

Utilities/

-ShadeRec.h/.cpp (Added variables: `Sync` variable, `Jump` and `Count` used for Samplers over multiple threads)

+Multithread.h (Multi-threading data structures)

+RandomNumber.h (Wrapper for random number generator)

+MTRand.h/.cpp (Mersenne twister random number generator)

+SmartPointer.h (`SmartPointer` Template)

+ReferenceCount.h (`ReferenceCount` class to be used by the `SmartPointer`)

Samplers/

-Sampler.h/.cpp (Function changes for multi-threading and random number generator)

Cameras/

-Camera.h/.cpp (Added overloaded function for multi-threading)

-Pinhole.h/.cpp (As above)

Tracers/

-Tracer.h/cpp (Added overloaded function for multi-threading)

-Whitted.h/cpp (Added overloaded function for multi-threading)

-AreaLighting.h/cpp (Added overloaded function for multi-threading)

-RayCast.h/cpp (Added overloaded function for multi-threading)