Multithreaded Ray Tracing from the Ground up Skeleton Framework

Download this multithreaded 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/

Original Skeleton made 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

Multithreading

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

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

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

The threading system uses wxThreads, a part of wxWidgets API. Each thread, 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 and thus will render the image a lot quicker than usual.

Reference counting with smart pointers

The multithreaded skeleton also uses SmartPointers 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.

Mersenne twister

For random numbers, the skeleton now uses a Pseudorandom Number Generator called Mersenne Twister, which provides fast generation of very high-quality pseudorandom numbers.

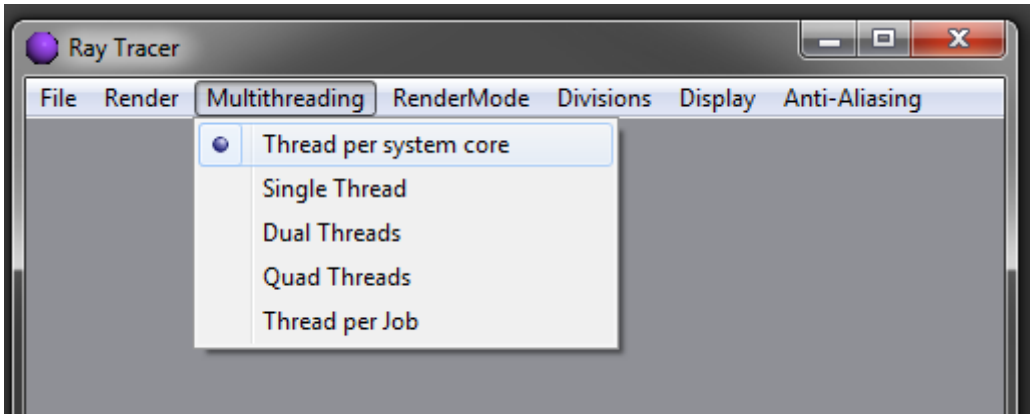
The Mersenne Twister provides a system independent random number generator that is portable across multiple operating systems, allowing for a random number seed to generate the same result on different computers and operating systems.

Optionally this can be easily changed to the system Rand through a Boolean value passed to the object, or set in the random number class.

Rendering options

There have been significant changes to the Interface of the multithreaded skeleton as described below.

Multithreading Options

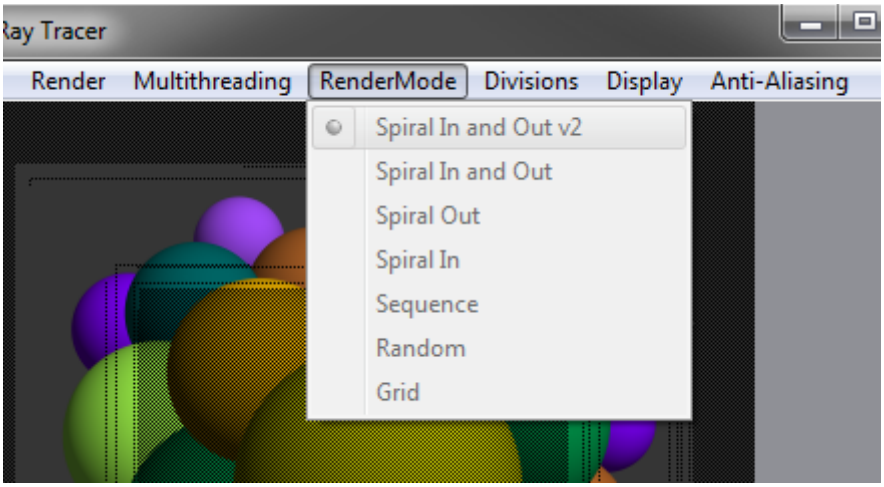


Under the Multithreading interface options you can choose how many threads you want to be involved in the rendering. By default a Thread per system core will be selected, the program will find out how many cores you have on your system and use that number of threads, including hyper threaded cores.

The other options are fairly self-explanatory, single thread will only use one thread, Dual for two, Quad for four.

Thread per Job is an experimental function that will create a thread per how many jobs there are. This will cause a lot of timesharing on your CPU and potentially a lot of deadlocks, not the most optimal way to render, however can have some nice visual effects.

RenderMode Options



Under the RenderMode interface options you can select the way in which the program will split up the jobs, which will give many different results (however the final image will 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 visual complete image at 50% render. This can be very helpful to see what the image will look like. It will then further fill in the final 50% to complete the render.

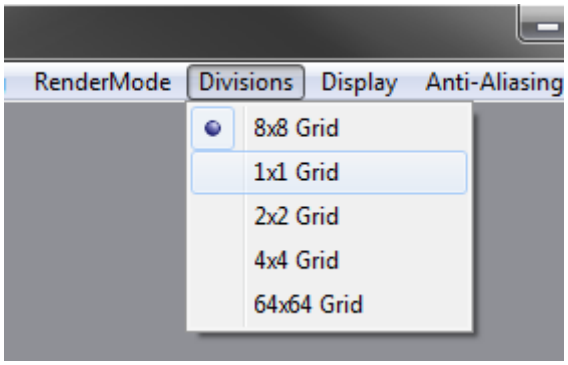
Spiral in and out will do a full render of outside and in (same as v2 just doesn't skip any pixels on the first half). Spiral Out will render from the centre outwards. Spiral In will render from the Outside in.

Sequence will setup the jobs in X (across) then Y (up) and will split them by divisions.

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

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

Divisions Options



Under the Divisions interface you can choose how many jobs there will be by how the pixels are divided up. A 8x8 division is the default and will give a fairly even break up of pixels for jobs.

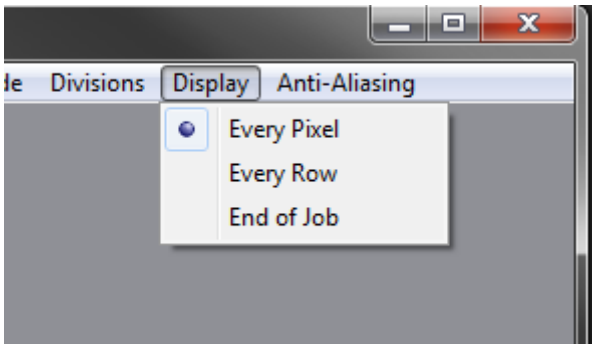
A 1x1 grid will set 1 job for the total of all pixels. This is only useful if you are using a Single core and do not want any other jobs. I would still recommend using 8x8 for a single core unless you are wanting to completely simulate the old skeleton.

A 2x2 grid 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 a Quad Core.

4x4 adds more jobs and is recommended if you don't want to use the 8x8.

64x64 is an experimental setting which will create 4096 jobs in total. This mode gives very interesting visual rending results but I definitely don't recommend using it all the time.

Display Options



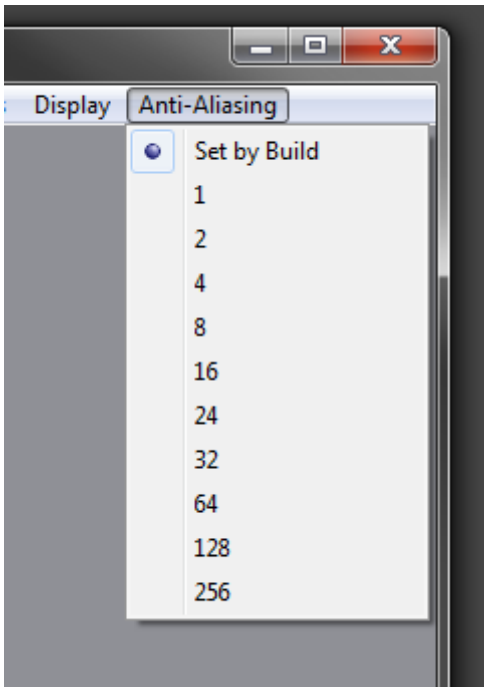
Under the Display interface drop down you can select how often you want the rendering threads to send their pixels back to the interface. This can be used to limit deadlocks between the interface thread

accepting pixels, which can be very helpful on very fast renders which may lock up the interface on fast computers.

The default setting is every pixel like the original skeleton. 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 modify this interface.

Anti-Aliasing Options

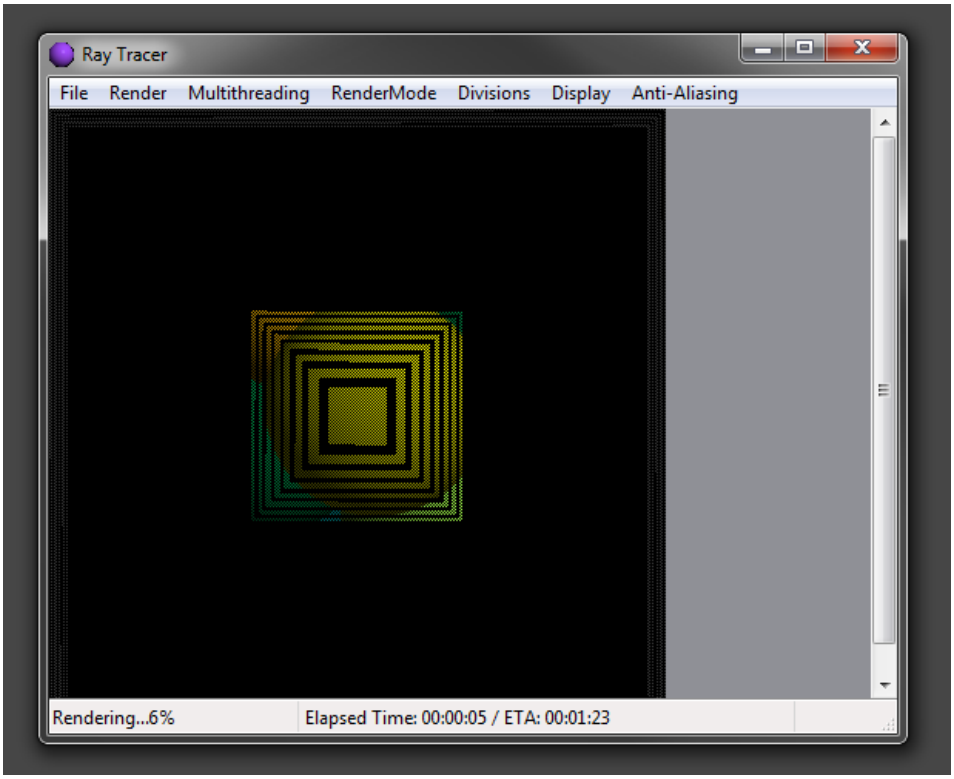


Under the Anti-Aliasing interface drop down you can set and override the antialiasing set by the build function. I added this as many times I was finding myself rebuilding and compiling just to change the sample size for the anti-aliasing.

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 setting the setting 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 down the bottom left. Also information about what stage in the render will also be shown here, such as building world, cleaning up.

Elapsed time and estimated time will be shown in the middle tab down the bottom.

File output options

The new skeleton will also allow you to output the following image file types:

.png, .tiff, .jpeg, .bmp

The JPEG will be saved with a quality of 100 (however it is recommended to save as a .png or tiff to conserve the proper quality.)

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, speed increase of 2-4 times in Release.
- Start your project with this Skeleton as the base if you want Multithreading, else as stated below you can merge them later.

Other Enhancements in the Multithreaded Skeleton:

- Multithreaded 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 Multithreaded Skeleton
2. Use the Multithreaded Solution as the main project and start adding your custom made classes and files 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. Note the implementation of this function with the skeleton is the basic type; you need to update the Sampler and other variables.
5. Make sure if you are using any compound objects to read the Notes below:

Notes on implementation

Implementing Grids or Compound objects you are going to want to be careful with the following function.

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

Materials cannot be returned the way described in the textbook for grids/compound objects (using a temporary local variable bound to the upper abstract object as this object is being used by multiple threads)

In the Multithreaded version you store the returned material inside the ShadeRec. "sr.material_ptr" instead of in the compound object material_ptr.

Changes from the Original Skeleton in Source Code:

User Interface/

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

World/

-World.h/cpp (Changes for Multithreading)

Utilities/

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

+MultiThread.h (Multithreading Data structures)

+RandomNumber.h (Wrapper for RandomNumber 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 Multithreading and RandomNumber generator)

Cameras/

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

-Pinhole.h/cpp (As above)

Tracers/

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

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

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

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