KIT308 Multicore Architecture and Programming

Assignment 2 — SIMD

Aims of the assignment

The purpose of this assignment is to give you experience at writing a program using SIMD programming techniques. This assignment will give you an opportunity to demonstrate your understanding of:

- the where there is one there is … (WTIOTIM);
- structures of arrays;
- the use of SIMD for calculation…
- the translation of conditional s…

Due Date

11:55pm Friday 23th of Septe… …ester)

- Late assignments will only be … …umstances and provided that the proper procedures have been followed (see the School Office or [this link](#) for de… …e submitted late without good reason will be subject to mark penalties if they are accepted at all (see the School office or [this link](#) for details on this as well).
- Forms to request extensions of time to submit assignments are available from the Discipline of ICT office. Requests must be accompanied by suitable documentation and should be submitted **before the assignment due date**.

Assignment Submission

Your assignment is to be submitted electronically via MyLO and should contain:

- A .zip (or .rar) containing a Visual Studio Solution containing a project for each attempted stage of the assignment (in the format provided in the downloadable materials provided below).
- A document containing:
  - A table of timing information comparing the original single-threaded code (from assignment 1), the provided base code times, and the Stage 1, 2, 3, 4, and 5 SIMD implementation on each scene file (all running with the maximum threads natively supported by your CPU).
  - An analysis of the above timing data.
- **You do not need to (and shouldn't) submit executables, temporary object files, or images.** In particular, you **must** delete the ".vs" diretory before submission as it just Visual Studio temporary files and 100s of MBs. Do not however delete the "Scenes" folder or the "Outputs" folder (but do delete the images within this one).

Marking

This assignment will be marked out of 100 (NOTE: there are 110 marks available, but it's only possible to receive a maximum of 100). NOTE: if your code for a particular stage does not correctly produce the expected output images, you will only be able to receive a maximum of half marks for that stage — see below for more details.

The following is the breakdown of marks:

Task/Topic	Marks
1. Conversion of Distance Calculation to Where There is One there is Many (WTIOTIM)	**15%**
Correct implementation of WTIOTIM [illegible]on (in *Distance.h*) to calculate distance to nearest object — i.e. manually inline `sphereDist`, `plan`[illegible]*FPrimitives.h*)	5%
Correct implementation of WTIOTIM [illegible]ex function — i.e. as above, but also manually inline `min` function (from *DFPrimitives.h*)	5%
Calculation of distance to centre poi[illegible]ore the `switch` statement (requires understanding the `union` used for the `SceneObject` datastruct[illegible]	5%
[illegible]ene Objects to SoA	**15%**
Correct declarations of data structures for SoA SIMD form of Scene Object container data (don't delete the AoS versions), and SoA SIMD form of data uses minimal (sensible) amount of memory	5%
Correct code to convert AoS container to dynamically declared SoA structures (conversion should happen after the scene has been loaded)	5%
Correct conversion of `distance` and `distanceAndIndex` to use SoA copy of Scene Objects	5%
3. AVX SIMD Conversion of Distance Calculation	**30%**
A. SIMD Conversion of `distance` function	**20%**
Correct and efficient (e.g. no use of if-statements, for-loops, or scalar code) calculation of distances to spheres, planes, and boxes, and correct declaration of loop constants before loop	5%
Correct and efficient (e.g. no use of switch-statements, if-statements, or scalar code) selection of the correct distance given the object type	5%
Correct and efficient (e.g. no use of if-statements, for-loops, or scalar code) calculation of final (scalar) minimum	5%
Correct handling of object list length not being divisible by 8	5%
B. SIMD Conversion of `distanceAndIndex` function	**10%**
Correct and efficient (e.g. no use of if-statements, for-loops, or scalar code) calculation of index (corresponding to current minimums)	5%
Correct (i.e. corresponding to scalar minimum) and efficient (e.g. no use of if-statements, for-loops, or scalar code) calculation of final (scalar) object index	5%
4. AVX SIMD Conversion of Rendering	**20%**
Correct and efficient SIMD conversion of `renderSection` to calculate 8 values at a time	5%
Correct and efficient SIMD conversion of `traceRay` to have SIMD arguments (and return values), and it should calculate 8 values at a time	5%
Correct and efficient SIMD conversion of `marchRay` to have SIMD arguments (and return values), and it should calculate 8 values at a time	5%
Correct and efficient SIMD conversion of `distanceAndIndex` to have SIMD arguments (and return values), and it should calculate 8 values at a time	5%
5. AVX SIMD Conversion of Lighting	**20%**
Correct and efficient SIMD conversion of `calculateIntersectionResponse`, `calculateNormal`, and `distance` to calculate 8 values at a time	5%
Correct and efficient SIMD conversion of `applyLighting` to calculate 8 values at a time	5%
Correct and efficient SIMD conversion of `applySpecular`, `applySoftShadow`, and `applyAmbientOcclusion` to calculate 8 values at a time	5%
Correct and efficient SIMD conversion of `applyDiffuse`, `applyCheckerboard`, `applyCircles`, and `applyWood` to calculate 8 values at a time	5%
Documentation	**10%**
Outputs showing timing information for the base assignment code and Stages 1–5 (with the maximum threads natively supported by your CPU) on all applicable scene files	5%
Analysis of data (comparisons across the base code and Stages 1–5)	5%
Penalties	
Failure to comply with submission instructions (eg. no cover sheet, incorrect submission of files, abnormal solution/project structure, etc.)	-10%
Poor programming style (eg. insufficient / poor comments, poor variable names, poor indenting, obfuscated code without documentation, compiler warnings, etc.)	up to -20%
Lateness (-20% for up to 24 hours, -50% for up to 7 days, -100% after 7 days)	up to -100%
Failure to use the techniques outlined in this unit's lectures and tutorials, resulting in a wildly different solution, using un-taught libraries (eg. an external library for SIMD, etc.), un-taught techniques, or a vastly different rendering implementation	up to -100%

Marking and Correct Images

As SIMD code is very *very* difficult to debug — as grows exponentially harder as the amount of it grows — there will be limited opportunity in the marking process to determine where exactly mistakes have been made, and even less chance of being able to provide fixes to those mistakes.

As a result, if your code for a stage does not produce the expected image, then you will only be eligible for half marks for that stage of the assignment.

In order to work within this constraint, you should attempt translations into SIMD in very small steps (i.e. a single line at a time, or even a partial line at a time). The lectures and live-coding sessions will demonstrate this approach to SIMD translation. You will likely receive more marks for a partially complete SIMD translation than a fully complete translation that doesn't work 100%.

Programming Style

This assignment is not focussed on p [illegible] it is concerned with efficient code), but you should endeavour to follow good programming practices. You should, [illegible]

- comment your code;
- use sensible variables names;
- use correct and consistent ind[illegible]
- internally document (with com[illegible] decisions.

[NOTE: any examples in the provided assignment materials that don't live up to the above criteria, should be considered to be deliberate examples of what not to do and are provided to aid your learning ;P]

The Assignment Task

You are to modify the (square-based) multithreaded implementation of a "simple" raytracer from the first assignment to take advantage of SIMD instructions. This requires changes across multiple files.

There are two base projects provided:

- *RayMarcherAss2Simplified*: For use **only** with Stage 1–3.
- *RayMarcherAss2*: For use **only** with Stage 4–5.

From the provided simplified (square-based) multithreaded raytracer implementation (*RayMarcherAss2Simplified*), for Stages 1–3 of the assignment you will create multiple s[illegible]dify the implementation as follows:

1. Rewritten functions for the dis[illegible]-is-one-there-is-many approach.
2. Translation of array-of-structu[illegible]arrays (SoA) for the scene object container.
3. Optimisation of the distance te[illegible]MD.

For Stage 4–5, you will **not** follow on[illegible]t instead continue from the provided *RayMarcherAss2* project. These stages will modify this implementation as follow[illegible]

4. Optimisation of the rendering [illegible]nce functions using SIMD to calculate multiple pixel fragment values simultaneously.
5. Optimisation of the lighting fu[illegible]ate multiple pixel fragment values simultaneously.

Implementation — SIMD Distance (Stages 1–3)

The following section describes in detail the steps required to complete Stages 1–3 of the assignment. If you complete these step, only then should you attempt Stage 4 and 5 (as they are likely more difficult).

1. Where there is One there is Many

This stage involves rewriting the `distance` and `distanceAndIndex` functions (in *Distance.h*) so that they use the where there is one there is many paradigm.

In order to complete this step you will need to:

- Inline all functions that act on a single scene object, i.e. `sphereDist`, `planeDist`, and `boxDist` (from *DFPrimitives.h*).
  - For `distanceAndIndex`, you need to also inline the specialised `min` function (from *DFPrimitives.h*.
- Determine how to perform the common calculation of the distance from the current point to the centre point of a scene object before the `switch` statement (i.e. simplify and potentially optimise the code by avoiding repeating the same calculation).

2. Structures of Arrays

This stage involves modifying the Scene struct (in *Scene.h*) to store a duplicate of the scene object (i.e. spheres, planes, and boxes) data via structure of arrays rather than the currently existing array of structures (currently in the `objectContainer` variable).

In order to complete this step you will need to:

- Create a SoA copy of the data that is stored in `objectContainer` in the `Scene` struct.
  - NOTE: this means the program has two copies of the object data, the original one in AoS form and a second one in SoA form. As stages 1–3 of this assignment predominately only requires changes to the distance functions, the rest of the code will still use the original AoS version of the data.
- Fill this SoA copy of the object data *after* the `Scene` struct has been loaded (dynamically allocating memory for the SoA representation).
- Rewrite the `distance` and `distanceAndIndex` functions to make use of the SoA.

Hints:

- The best place to do the conversion is immediately *after* loading the scene (i.e. in *Raytrace.cpp*).
- You should not need to attempt to modify *Scene.cpp* to complete this step.
- `SceneObject` is defined in *SceneObjects.h*.
- `SceneObject` uses a `union` to overlay data for spheres, planes, and boxes. It's up to you whether you attempt to replicate this structure somehow with your SoA. Doing this successfully is required for full marks on this step, but is not required to successfully move on to and complete Stage 3.
- At times this code update may require conversions to and/or from normal structs (such as `Vector` and `Point`) to the equivalently stored data in the SoA.
- It is up to you whether your SoA datastructure uses SIMD datatypes or not.

At the end of this stage the program should still produce the same results as the base code. Be thorough in your testing (i.e. test all the appropriate scenes) to ensure that everything works correctly before progressing.

3A. SIMD Conversion of `distance` function

This stage involves converting the first of the distance functions from Stage 2 into an AVX SIMD implementation.

In order to complete this step you will need to:

- Converting the input parameter `currentPoint` into a AVX SIMD-ready value (or values).
- Step through the SoA object container in chunks proportional to the number of value stored in each AVX SIMD variable (i.e. 8).
- Convert the calculation to use SIMD. This will require calculating multiple minimum distances (one for each SIMD lane).
- Take care to correctly deal with any conditional expressions in the loop. i.e. the switch-statement must be converted to SIMD code via the use of appropriate masking statements.
- Consolidation of the values calculated using SIMD into a single scalar return value. This should be done after the loop finishes.
- Ensure your approach deals with the situation where the object count isn't equally divisible by the number in each SIMD calculation (e.g. a scene with 9 planes with 8 in each SIMD value). Doing this correctly either requires special masking code, or for you to revisit Stage 2.

3B. SIMD Conversion of `distanceAndIndex` function

This stage involves converting the latter of the two new distance functions from Stage 2 into an AVX SIMD implementation.

Most of this conversion is identical to step 3A, and the code can just be copied from that function. It does however require the addition of:

- New variables to keep track of the indices of the objects that are currently closest for each SIMD lane.
- Consolidation of the index values calculated using SIMD into a single scalar index value. This should be done after the loop finishes.

Hints:

- You do not want to return the *minimum* index value (i.e. the smallest of all the indices in a SIMD vector), you want to return the index corresponding to the minimum distance found (i.e. the index in the same lane as the minimum distance).

General Hints / Tips

- The techniques required to complete each stage rely heavily on work done in the SIMD tutorials and the step-by-step approach shown in lectures — refer to them often
- When implementing the SIMD … small portions of code at a time (e.g. even a single line is often a lot) and then perform the rest of the calcula… lar code, or even compare the result of the SIMD version to the existing scalar code.
- Again for SIMD, it's helpful to … educed size for testing, with an equally small block size, one sample per pixel, and with one thread (e.g. try using … *ze 8 8 -blockSize 8 -samples 1 -threads 1*). It may even be helpful to fix the number of rays cast (`MAX_RAYS` … tion doesn't occur. This doesn't produce a very nice image, but images can be easily compared visually for pr… 64 pixels) and `printf` statements can be used to verify the correctness of calculations without outputting … use of debuggers is somewhat perilous in a multithreaded environment, but that shouldn't be an issue here).
- When converting conditional s… helpful to do this with scalars first (although remember that C/C++ implementations use `1` rather … the calculations may be different).
- Write functions to output all the elements in a SIMD value for easier `printf` debugging (I am not a fan of how these types are shown in the debugger — unless they have changed recently).

Implementation — SIMD Rendering and Lighting (Stages 4–5)

The following section describes in detail the steps required to complete Stage 4–5 of the assignment. You should only attempt this step if you have fully completed (and tested) Stages 1–3 (as these stages are likely *much* harder to complete).

Stages 1–3 optimised the distance calculation, but it's difficult to see how to further optimise the code after Stage 3. It has the following problems:

- The calling sites of `distance` and `distanceAndIndex` can't obviously be turned into SIMD code.
- The code had to use a simplified renderer that only has the "Union" operation (and not "Intersection", "Difference", or "Smooth"). This is because it's not possible to apply different operations between elements in a SIMD vector in an efficient way.

Stages 1–3 were effectively optimising the code from the inside-out (i.e. starting at the innermost loop/function, and working outwards). This is normally the most effective approach, and certainly the quickest way to get SIMD performance. In Stages 4–5, you are going to optimise the code from the outside-in.

NOTE: Stage 4 does not build upon your solution to Stage 3 — instead you should copy the files from the *RayMarcherAss2* project to begin.

4. SIMD Rendering

This stage involves rewriting some code from the `renderSection` function, and then all the way down through the code until rewriting the distance functions again. With this approach we are attempting to calculate multiple pixel (fragment) values at once.

In order to complete this step you will need to:

- Rewrite `renderSection` to calculate 8 pixel (fragment) values at a time.
  - NOTE: this step can still be considered completed with `traceRay` retaining its original arguments and being called by a scalar for-loop, or have SIMD arguments and contain an additional for-loop internally — until you complete the following step.
- Rewrite `traceRay` to have SIMD arguments and calculate 8 pixel (fragment) values at a time.
  - NOTE: this step can still be considered completed with `marchRay`, `calculateIntersectionResponse`, and `applyLighting` retaining their original arguments and being called by a scalar for-loop, or have SIMD arguments and contain an additional for-loop internally — until you complete the following step.
- Rewrite `marchRay` to have SIMD arguments and calculate 8 pixel (fragment) values at a time.
  - NOTE: this step can still be considered completed with `distanceAndIndex` retaining its original arguments and being called by a scalar for-loop, or have SIMD arguments and contain an additional for-loop internally — until you complete the following step.
- Rewrite `distanceAndIndex` to have SIMD arguments and calculate 8 pixel (fragment) values at a time.
  - NOTE: all subfunctions need to be converted to SIMD also.

5. SIMD Lighting

This stage involves rewriting the remaining lighting code from the `traceRay` function, and then all the way down through the code until the final tecturing functions. With this approach we are attempting to calculate multiple pixel (fragment) values at once.

In order to complete this step you will need to:

- Rewrite `calculateIntersectionResponse`, `calculateNormal`, and `distance` to calculate 8 pixel (fragment) values at a time.
- Rewrite `applyLighting` to calculate 8 pixel (fragment) values at a time.
  - NOTE: this step can still be considered completed with `applyDiffuse`, `applySpecular`, `applySoftShadow`, and `applyAmbientOcclusion` retaining its original arguments and being called by a scalar for-loop, or have SIMD arguments and contain an additional for-loop internally — until you complete the following steps.
- Rewrite `applySpecular`, `applySoftShadow`, and `applyAmbientOcclusion` to calculate 8 pixel (fragment) values at a time.
- Rewrite `applyDiffuse`, `applyCheckerboard`, `applyCircles`, and `applyWood` to calculate 8 pixel (fragment) values at a time.

Hints/Tips:

- Remember to translate code in small steps.
- You have to change complex conditional control structures (e.g. continues, switch statements, etc.) into SIMD code. It's easier to do this using scalar code first, and then check your work, before attempting SIMD conversion.

Missing Mathematical Functions and Helper Classes

There are no definitions in the AVX/AVX2 instruction sets of some mathematical functions needed to translate some of this code to SIMD. You will need to write your own using SIMD intrinsics.

You might find it helpful to write extra helper structs like `Vector8` (e.g. `Colour8`, `DistanceAndIndex8` etc.), and also to add helper functions to these structs.

Hints / Tips

- All the hints/tips from the stages 1–3 apply here (i.e. complete this in small steps, use 8x8 images as tests, use a lot of `printfs`, etc.).
- It's up to you to decide if there is any benefit to a SoA approach for any data for these Stages.
- These stages are much more work than Stages 1–3 and they are worth less marks. Decide if you think it's worth it.

Documentation

When completing either stage of the assignment you should provide:

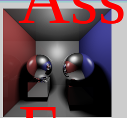
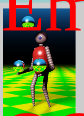
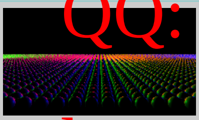
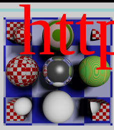
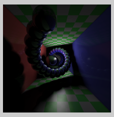
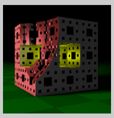
- timing information for each scene file for the average time (to 1 decimal place) taken over 5 runs for a render using a thread count supported by your the number of logical processes in your system and a block size of 16 (see a later section for an example format for this timing table; and
- an explanation of the results (e.g. why there's no difference between the performance of stages 1, 2, and 3 (*NOTE: this is a made up example and isn't necessarily what to expect*), or why a particular type of implementation works well (or poorly) on a particular scene, etc.). This explanation should be with respect to the CPU on the system on which you ran the tests, and you should discuss how the architectural features of the CPU explain th[illegible]

Tests / Timing

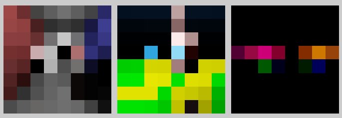
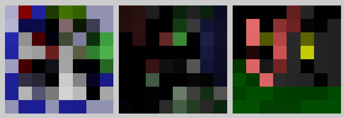
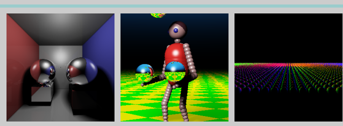
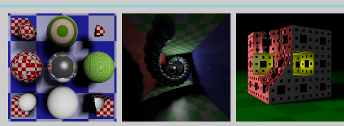
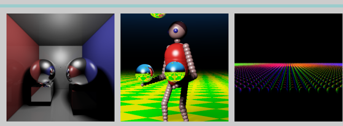
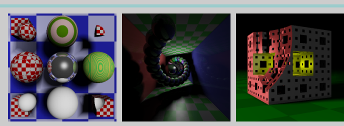
The following table lists all the tests [illegible] erate correctly at each stage. It also shows the timing tests that need to be performed in order to fully complete [illegible] f the assignment. Fully completing this tests may take up to an hour (with the 5 required runs) on some hardware, sc[illegible]

In order to confirm your images mat[illegible] e base version of the assignment code, it's strongly recommended you use an image comparison tool. For part of th[illegible] agick will be used (as it was in Assignment 1).

The timing tests need to be run in th[illegible] on as well as "FastRelease" build configuration. The "Release" mode uses a "strict" floating-point model setting to (attempt to) ensure that the SIMD conversion doesn't introduce calculation inconsistencies with the base code, and the "FastRelease" allows for inaccuracies in order to prioritise speed. Accordingly, only the former should be used for the comparison tests with ImageMagick (and its this version that will be used by the marker to perform these tests) — the latter will show a varying amount of difference depending on the test.

Timing Test			Build Config	Average Time of 5 Runs (Milliseconds)						
				Base Code	Simplified Base Code	Stage 1 WTIOTIM	Stage 2 SoA	Stage 3 SIMD Distance	Stage 4 SIMD Rendering	Stage 5 SIMD Lighting
1.	-input Scenes/cornell.txt -size 1024 1024 -samples 1		Release							
			FastRelease							
2.	-input Scenes/juggler.txt -size 720 1280 -samples 1		Release							
			FastRelease							
3.	-input Scenes/5000spheres.txt -size 240 135 -samples 1		Release							
			FastRelease							
4.	-input Scenes/all.txt -size 1024 1024 -samples 1		Release		x	x	x	x		
			FastRelease		x	x	x	x		
5.	-input Scenes/spiral.txt -size 1024 1024 -samples 1		Release		x	x	x	x		
			FastRelease		x	x	x	x		
6.	-input Scenes/sponge.txt -size 256 256 -samples 1		Release		x	x	x	x		
			FastRelease		x	x	x	x		

The following tests will be run on your code for each scene file. You also might find the 8x8 tests useful for your SIMD code conversion:

Test		Image Result	
		Stages 1–5	Stages 4–5
1.	-blockSize 8 -size 8 8 -samples 1 -threads 1		
2.	-blockSize 16 -size 256 256 -samples 1		
3.	-blockSize 16 -size 256 256 -samples 2		

Provided Materials

The materials provided with this assignment contain:

- The source code of the base multi-threaded version of the raytracer (i.e. a solution to the Stage 3 of Assignment 1).
- The source code of the base multi-threaded version of the raytracer with a simplified distance calculation that removes all but one object operator).
- A set of scene files to be supplied to the program.
- A set of reference images for testing.
- Some batch files for testing purposes.

[Download the materials as an ZIP file]

Source Code

The provided MSVC solution contains

RayMarcherAss2

The provided code consists of 22 sou

- **Raytracing logic:**
  - *Raytrace.cpp*: this file contains the `main` function which reads the supplied scene file, begins the raytracing, and writes the output BMP file.
  - *Intersection.h* and *Intersection.cpp*: these files define a datastructure for capturing relevant information at the point of intersection between a ray and a scene object, as well as the main ray trace function.
  - *DFPrimitives.h*: this header contains definitions for functions to determine the distance to primitive scene objects (i.e. spheres, planes, and boxes). It also provides functions for an object that contains a distance and an index.
  - *Distance.h*: this header contains definitions for functions to determine the distance to all objects in the scene. There are two versions of the distance function: one only returns the distance to the nearest object, the other returns the distance and the index of the nearest object.
  - *Lighting.h* and *Lighting.cpp*: these files provide functions to apply a lighting calculation at a single intersection point.
  - *Texturing.h* and *Texturing.cpp*: these files provide functions for the reading points from 3D procedural textures.
  - *Constants.h*: this header provide constant definitions used in the raytracing.
- **Basic types**:
  - *Primitives.h*: this header contains definitions for points, vector, and rays. It also provides functions and overloaded operators for performing calculations with vectors and points.
  - *SceneObjects.h*: this header file provides definitions for scene objects (i.e. materials, lights, spheres, planes, boxes, and the combined scene object).
  - *Colour.h*: this header defines a datastructure for representing colours (with each colour component represented as a `float`) and simple operations on colours, including conversions to/from the standard BGR pixel format.
- **SIMD Helpers**:
  - *PrimitivesSIMD.h*: this header contains operators for many common SIMD functions as well as a definition for `Vector8` (as a representation for 8 vectors). It also provides functions and overloaded operators for performing calculations with `Vector8`s.
- **Scene definition and I/O**:
  - *Scene.h* and *Scene.cpp*: the header file contains the datastructure to represent a scene and a single function that initialises this datastructure from a file. The scene datastructure itself consists of properties of the scene and lists of the various scene objects as described above. The implementation file contains many functions to aide in the scene loading process. Scene loading relies upon the functionality provided by the `Config` class.
  - *Config.h* and *Config.cpp*: this class provide facilities for parsing the scene file.
  - *SimpleString.h*: this is helper string class used by the `Config` class.
- **Image I/O**:
  - *ImageIO.h* and *ImageIO.cpp*: these files contain the definitions of functions to read and write BMP files.

RayMarcherAss2Simplified

The provided code consists of 22 source files.

There is just one modified file that differs from the above:

- **Raytracing logic**:
  - *Distance.h*: both versions of the distance function are modified to only have the "Union" operator, i.e. they only calculate the minimum distance to the nearest object with no boolean operators.

Stage1 – Stage5

These projects are empty.

To begin work on the assignment you should (in Windows Explorer) copy all of the 22 *.h* and *.cpp* files from *RayMarcherAss2Simplified* into the *Stage1* folder and then right-click on the *Stage 1* project in Visual Studio and choose "Add / Exiting Item..." and add those 22 files.

Executing

The program has the following functionality:

- By default it will attempt to load the scene "Scenes/cornell.txt" and render it at 1024x1024 with 1x1 samples (using the maximum number of threads supported by the CPU natively) with a block size of 16.
- By default it will output a file named "Outputs/[*scenefile-name*]_[*width*]x[*height*]x[*sample-level*]_[*executable-filename*].bmp" (e.g. with all the default options, "Outputs/cornell.txt_1024x1024x1_RayMarcherAss2.exe.bmp")
- It takes command line arguments that allow the user to specify the width and height, the anti-aliasing level (must be a power of two), the name of the source scene file, the name of the destination BMP file, and the number of times to perform the render (to improve the timing information).
- Additionally it accepts some ar[illegible]ber of threads, whether each thread will tint the area that it renders, whether each thread will instead colour[illegible]read as a solid grey, and the size of the block to render.
- It loads the specified scene.
- It renders the scene (as many[illegible]
- It produces timing information[illegible]to produce a render ignoring all file IO.
- It outputs the rendered scene[illegible]

For example, running the program at[illegible]arguments would perform the first test (as described in the scene file section):

On execution this would produce out[illegible]as well as writing the resultant BMP file to *Outputs/cornell.txt_1024x1024x1_R*[illegible]

```
rendered 1048576 samples using 8 threads, average time taken (1 run(s)): 12578.0ms
```

Testing Batch Files

A number of batch files are provided that are intended to be executed from the command line, e.g.

- For timing (using the "Release" build configuration):
  - `baseTiming.bat` will perform all the timing tests required for the base code (i.e. 5 runs with the appropriate amount of threads for each Test scene).
  - `simplifiedTiming.bat` will perform all the timing tests required for the simplified base code.
  - `stage1Timing.bat` will perform all the timing tests required for Stage 1.
  - `stage2Timing.bat` will perform all the timing tests required for Stage 2.
  - `stage3Timing.bat` will perform all the timing tests required for Stage 3.
  - `stage4Timing.bat` will perform all the timing tests required for Stage 4.
  - `stage5Timing.bat` will perform all the timing tests required for Stage 5.
- For timing (using the "FastRelease" build configuration):
  - `baseTiming2.bat`.
  - `simplifiedTiming2.bat`.
  - `stage1Timing2.bat`.
  - `stage2Timing2.bat`.
  - `stage3Timing2.bat`.
  - `stage4Timing2.bat`.
  - `stage5Timing2.bat`.
- For testing (requires Image Magick installation), e.g.:
  - `stage1Tests.bat` will perform all the comparisons required for Stage 1 Tests.
  - `stage2Tests.bat` will perform all the comparisons required for Stage 2 Tests.
  - `stage3Tests.bat` will perform all the comparisons required for Stage 3 Tests.
  - `stage4Tests.bat` will perform all the comparisons required for Stage 4 Tests.
  - `stage5Tests.bat` will perform all the comparisons required for Stage 5 Tests.