# EECS 147 SPRING 2023
# FINAL PROJECT:

# GPU-ACCELERATED
# OBJECT-ORIENTED RAY TRACER

---


**Team The Ray Tracers:**

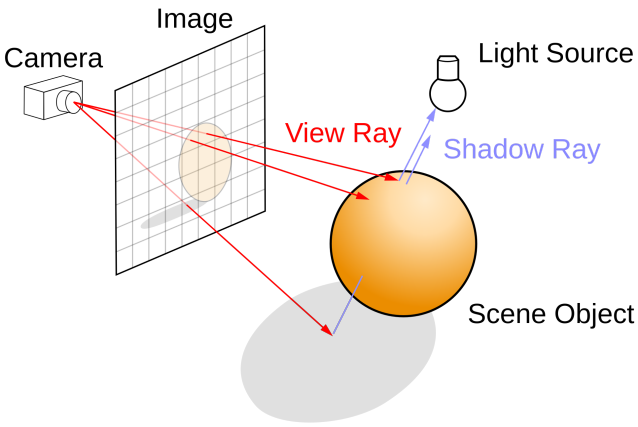
Tanisha Basrai, Justin Sanders, Vy Vo


**Parallel Ray Tracer Repository**

**Ray Tracer Presentation Video**

## Table Of Contents:

## Overview:




Ray tracing is a rendering technique that models the interaction between light and 3D objects in a scene. Ray tracing works by following a path from an imaginary camera through each pixel in the image and setting that pixel's color to the corresponding color of the object visible through it. The color of objects is calculated with respect to physical interactions with light sources around it.

An implementation of ray tracing is developed in UCR's CS130: Computer Graphics course. This is the source code that our project is based on.

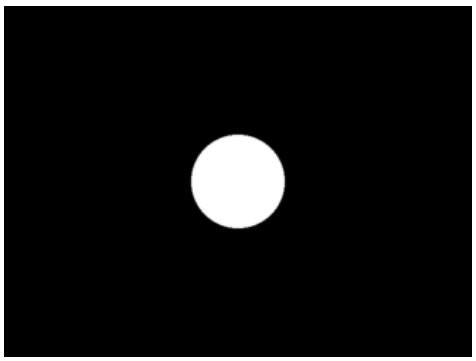
Our project objective is to run a pared down and CUDA friendly version of the Ray Tracer program developed in CS130 on the GPU and observe the difference in execution time compared to the CPU.

## Original Ray Tracer Implementation:

The original ray tracer is built using object-oriented C++. The following sequence is how an image is generated from the CS130 ray tracer:

1) Run the ray tracer with an input.txt file that defines (most importantly): image resolution, rgb color values, shader type with color, object type with location, shaded objects (the final object with location and color), and the camera object. The example .txt below is the test file to create a centered white sphere.

```
size 640 480 ———— resolution
color white 1 1 1
flat_shader white_shader white
sphere S 0 0 0 1
shaded_object S white_shader —— create white sphere
enable_shadows 0
recursion_depth_limit 1
camera 0 4 6 0 0 0 0 1 0 70
```



2) The values defined in the input.txt file are parsed by the `Parse` class who stores the objects and values in a `Render_World` object.
3) The "filled out" `Render_World` object, now with its mathematical world defined, loops over every pixel in the image and renders it.
4) For each pixel, a "ray" is cast from the camera's position to that pixel. Based on the intersection of that ray with shaded objects in the world, a color for that pixel is set and saved to an output matrix.

a) The color of objects (and therefore pixels) is determined by how they are shaded. Below is an example of how different shaders affect object color.

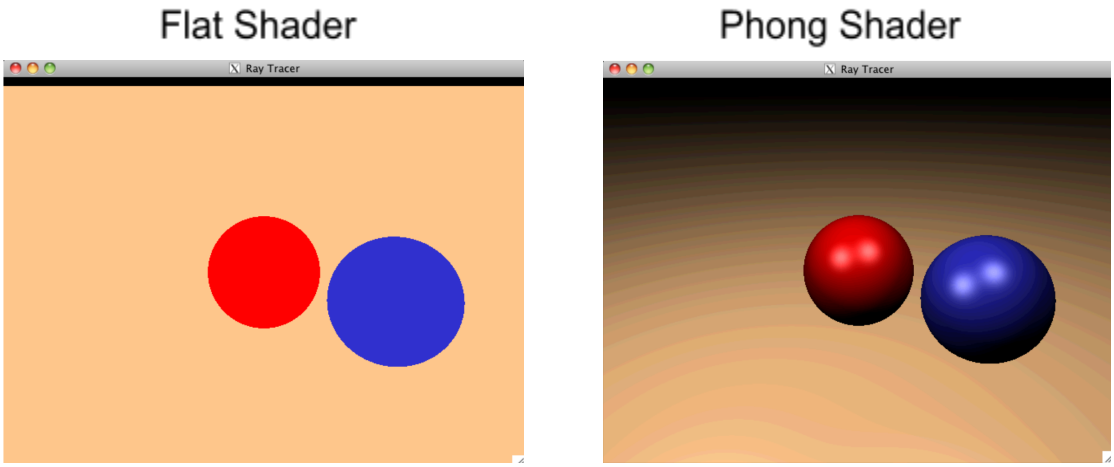



*Figure 1*

In short, the ray tracer is built to store objects from an input .txt file, render those objects by using the ray tracing technique, and output the final image (the scene from the perspective of the imaginary camera).

## GPU Utilization:

To parallelize this computation, nested for-loops are replaced with a kernel launch that has a grid set by the ceiling of the image dimensions divided by the size of our thread block. For example an image of size 100x160 pixels using 16x16 thread blocks will be broken down into a grid with 7x10 thread blocks.

Within the kernel, each thread is assigned to an output pixel position (at position x, y of the image), if the pixel position is within the boundaries of the image size, the thread calls the `Render_Pixel()` function. The actual process for an individual pixel to be computed is unchanged, but the GPU accelerates the program by computing hundreds of pixels simultaneously.

## Getting It to Work (Implementation Phases):

### Phase 1: Converting C++ Classes and Structures to Unified Memory

We will need to add all of the necessary classes and structures to unified memory so that these classes and structures could be accessed and modified by both the CPU and GPU. Without unified memory, we will have to allocate memory on both the CPU and GPU, copy the data from CPU to GPU, launch and calculate for the result in the kernel, and then copy the result from the GPU back to the CPU. Using unified memory allocation will allow us to create memory for the objects on both the CPU and GPU and be able to make changes to the data without memory copying between the CPU and GPU.

To add classes and structures to unified memory, we will need to create a managed class that overloads new and delete. This allows us to allocate the memory for both CPU and GPU to use and delete from unified memory when we create new objects and delete those objects.Any classes and structures that inherits the Managed class, inherits the new and delete operator.

When using unified memory, we will need to make deep copies, especially any classes or structures that have pointers. We will need to allocate memory for those pointers.

Add `__host__ __device__` to all member functions of classes and structures in unified memory. This allows the member functions to be used on both the host and device side.

### **Phase 2:** Convert Object Vectors to Arrays

STL cannot be used inside kernels. A solution to this is to convert the vectors used in our classes to arrays with a defined `ARRAY_SIZE`.

### **Phase 3:** Remove Abstract Classes and Modify Parse

Abstract Class objects cannot be created on the host side and passed to the device side, "it is not allowed to pass as an argument to a `__global__` function an object of a class derived from virtual base classes." (NVIDIA).

The source code is highly dependent on polymorphism, with *Shader* and *Object* classes being the backbone of rendering objects and object colors, and because of this, we played with a number of different solutions.

One solution was to split the initial parsing and object creation into two parts. First, parse through the .txt world building data and store the parse information in a `string` struct. Second, when the kernel is launched, pass this parse struct to the device and create the abstract class objects in the kernel using the `string` struct to aid with object creation. However, with how complex the world is (the colors, objects, lights, shaders, and shaded objects), allocating the correct memory size was difficult to plan out. In addition to this, it gets a bit confusing when we are mixing unified memory and gpu memory allocation. Our other major issue with this is that the computation itself was done in `render_world.cu` and we would need to pass the render_world object with allocated but empty class objects arrays. Overall, since there were not many resources and examples with CUDA polymorphism and given the time constraint of the project, we did not commit to this idea.

Our final solution was to avoid polymorphism altogether. We converted all abstracted classes into concrete classes and modified any reference to those abstract classes. This meant changing classes, functions, member variables, as well as making a slew of other modifications in the Parsing class. Our final un-polymorphized product can be seen in Figure 2, and more implementation details can be found in the Problems Faced: Polymorphism section of this report.

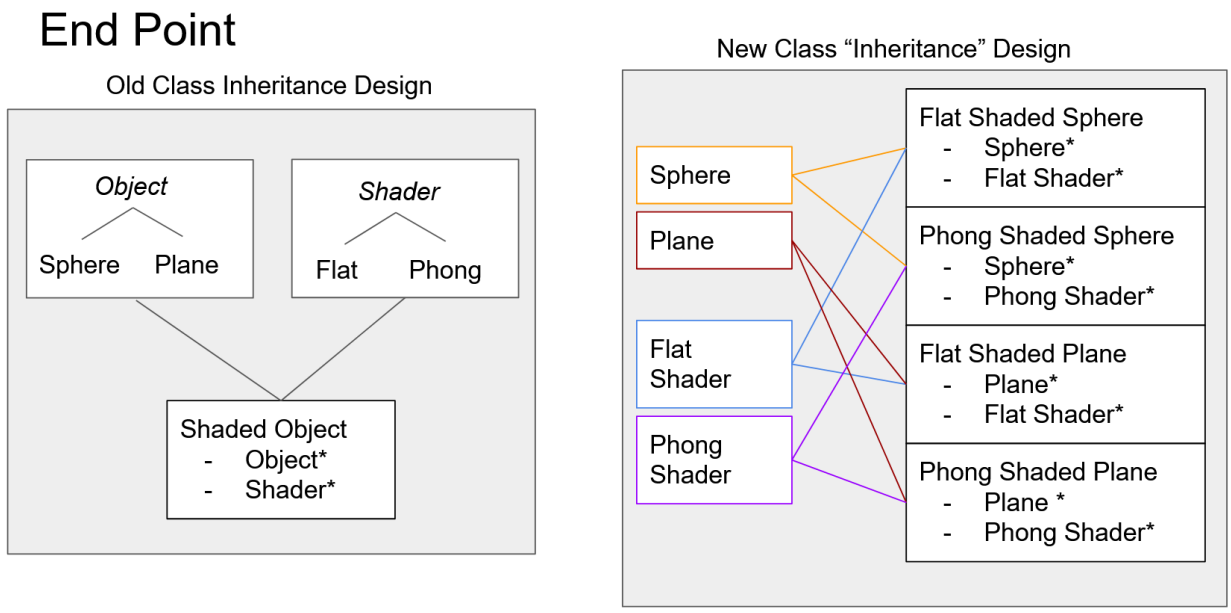



*Figure 2*

## **Phase 4:** Launch Kernel and Render

For the kernel launch, we define the thread blocks to be squares of 16x16 and a grid of thread blocks large enough to fully cover the test image. This may include some overhang on the edges of the image, but this will be handled by bound checking within the kernel itself.

```
int col = threadIdx.x+blockDim.x*blockIdx.x;
int row = threadIdx.y+blockDim.y*blockIdx.y;

//camera_num_pixels[1] = col width, camera_num_pixels[0] = row width
if(col < r->camera->number_pixels[0] && row <
r->camera->number_pixels[1])
{
    r->Render_Pixel(ivec2(col, row));

}
__syncthreads();
```

## How to Run Instructions:

**To run the repository the environment needs the following:**

- C++ 17 support
- CUDA compatible GPU
- CUDA SDK
- NVC++
- UNIX compatibility (Linux or similar as program depends on “unistd.h” methods)
- Libpng for C++

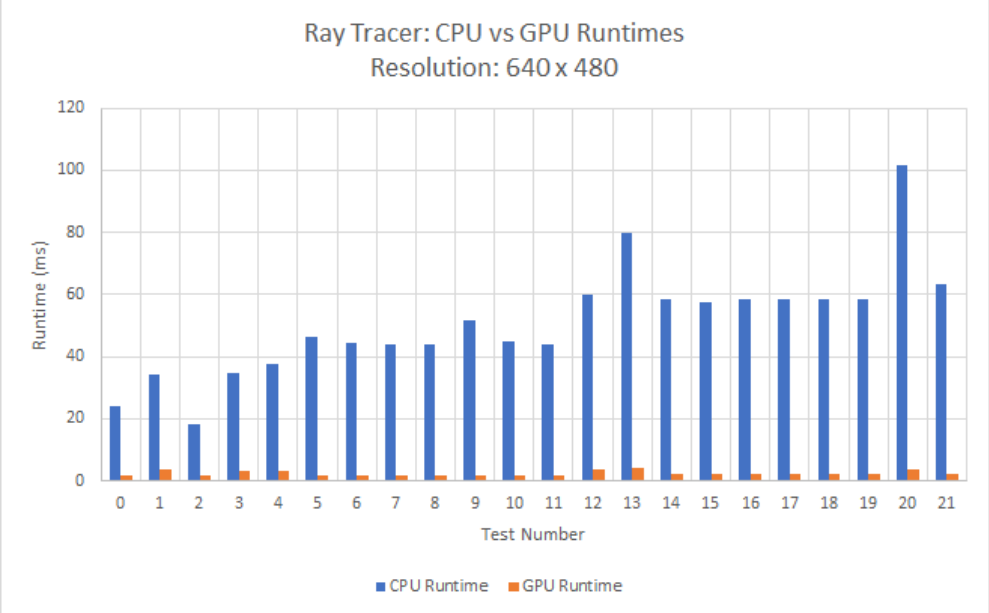
We developed and tested on the UCR Bender Server.

**Steps to Run:**

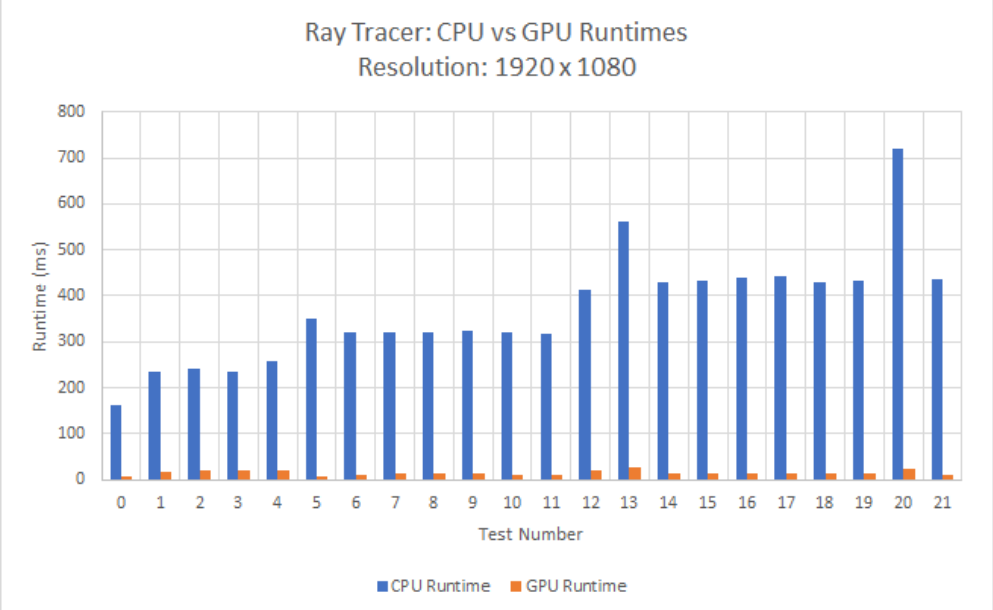
- Clone Repo to a directory
  - `git clone`
- If running on Bender all environmental dependencies are installed, but if that changes the singularity shell has the needed installs. Shell **MAY** need to be launched from the home directory.
  - `singularity shell --nv /singularity/cs217/cs217.sif`
    - This file is NOT included in the repo as it was made available, but not provided to us directly
- Navigate to Repo and compile
  - `nvc++ -std=c++17 -O3 *.cpp *.cu -o ray_tracer -lpng -lcudart -lcuda`
- Run the Ray Tracer
  - Test Files are located in the tests/ subdirectory.
  - Input files are .txt
  - Expected output images are .png
  - Run Test and have output stored as output.png
    - `./ray_tracer -i {test_directory_name/test}.txt`
    - `./ray_tracer -i tests-cpu/00.txt`
- Run Test and have output compared against expected output (Error is both printed as a value to the terminal and stored as an image labeled diff.png)
  - `./ray_tracer -i {test_directory_name/test_file} -s {test_directory_name/expected_output_image}`
  - `./ray_tracer -i tests-cpu/00.txt -s tests-cpu/00.png`
- Runtime Statistics will print to the terminal.
  - Open the generated png files to visually inspect results.

## Results:

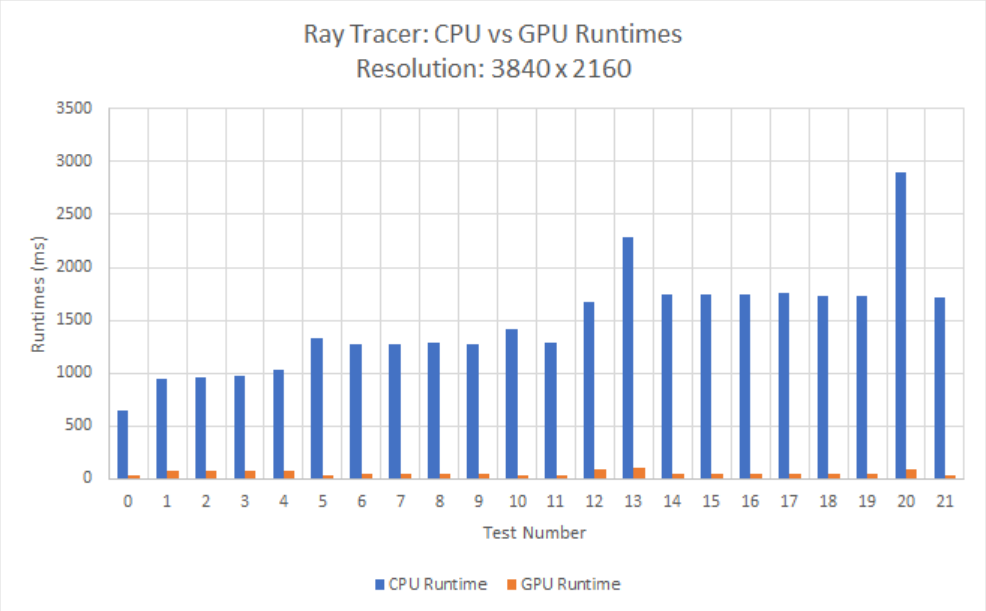
| Resolution = 640 x 480 | | | |
|---|---|---|---|
| **Test Number** | **CPU Runtime (ms)** | **GPU Runtime (ms)** | **Reduction Percentage (%)** |
| Flat Shader Tests | | | |
| 0 | 23.78 | 1.588 | 93.32% |
| 1 | 34.337 | 3.555 | 89.65% |
| 2 | 18.043 | 1.838 | 89.81% |
| 3 | 34.644 | 3.154 | 90.90% |
| 4 | 37.682 | 3.265 | 91.34% |
| Phong Shader Tests | | | |
| 5 | 46.507 | 1.576 | 96.61% |
| 6 | 44.128 | 1.834 | 95.84% |
| 7 | 43.852 | 1.746 | 96.02% |
| 8 | 43.749 | 1.787 | 95.92% |
| 9 | 51.71 | 1.819 | 96.48% |
| 10 | 44.94 | 1.768 | 96.07% |
| 11 | 43.906 | 1.797 | 95.91% |
| 12 | 59.768 | 3.771 | 93.69% |
| 13 | 79.735 | 4.125 | 94.83% |
| 14 | 58.579 | 2.043 | 96.51% |
| 15 | 57.684 | 2.081 | 96.39% |
| 16 | 58.387 | 2.14 | 96.33% |
| 17 | 58.611 | 1.997 | 96.59% |
| 18 | 58.351 | 2.067 | 96.46% |
| 19 | 58.333 | 2.076 | 96.44% |
| 20 | 101.682 | 3.705 | 96.36% |
| 21 | 63.37 | 1.95 | 96.92% |
| AVERAGE | | | 94.93% |



| Resolution = 1920 x 1080 | | | |
|---|---|---|---|
| Test Number | CPU Runtime (ms) | GPU Runtime (ms) | Reduction Percentage (%) |
| Flat Shader Tests | | | |
| 0 | 160.86 | 7.105 | 95.58% |
| 1 | 234.332 | 18.793 | 91.98% |
| 2 | 241.394 | 19.427 | 91.95% |
| 3 | 234.766 | 19.544 | 91.68% |
| 4 | 256.567 | 20.191 | 92.13% |
| Phong Shader Tests | | | |
| 5 | 352.066 | 7.983 | 97.73% |
| 6 | 319.426 | 9.841 | 96.92% |
| 7 | 320.852 | 12.808 | 96.01% |
| 8 | 320.576 | 12.578 | 96.08% |
| 9 | 322.731 | 12.517 | 96.12% |
| 10 | 320.597 | 12.162 | 96.21% |
| 11 | 319.105 | 9.109 | 97.15% |
| 12 | 413.593 | 19.138 | 95.37% |
| 13 | 562.729 | 25.63 | 95.45% |
| 14 | 430.489 | 12.683 | 97.05% |
| 15 | 433.421 | 12.569 | 97.10% |
| 16 | 440.437 | 13.059 | 97.03% |
| 17 | 442.385 | 12.948 | 97.07% |
| 18 | 430.091 | 12.855 | 97.01% |
| 19 | 431.845 | 12.712 | 97.06% |
| 20 | 720.964 | 24.666 | 96.58% |
| 21 | 435.548 | 10.147 | 97.67% |
| AVERAGE | | | 95.77% |



| Resolution = 3840 x 2160 | | | |
|---|---|---|---|
| **Test Number** | **CPU Runtime (ms)** | **GPU Runtime (ms)** | **Reduction Percentage (%)** |
| Flat Shader Tests | | | |
| 0 | 644.94 | 28.495 | 95.58% |
| 1 | 948.066 | 75.005 | 92.09% |
| 2 | 966.592 | 76.642 | 92.07% |
| 3 | 970.983 | 75.241 | 92.25% |
| 4 | 1026.005 | 78.347 | 92.36% |
| Phong Shader Tests | | | |
| 5 | 1329.035 | 30.049 | 97.74% |
| 6 | 1268.494 | 40.051 | 96.84% |
| 7 | 1280.002 | 39.867 | 96.89% |
| 8 | 1284.98 | 39.895 | 96.90% |
| 9 | 1276.5 | 40.543 | 96.82% |
| 10 | 1408.81 | 39.837 | 97.17% |
| 11 | 1281.01 | 39.841 | 96.89% |
| 12 | 1672.525 | 89.814 | 94.63% |
| 13 | 2283.264 | 101.051 | 95.57% |
| 14 | 1738.993 | 48.916 | 97.19% |
| 15 | 1749.592 | 49.314 | 97.18% |
| 16 | 1745.347 | 49.659 | 97.15% |
| 17 | 1754.105 | 50.117 | 97.14% |
| 18 | 1734.139 | 49.153 | 97.17% |
| 19 | 1722.894 | 49.963 | 97.10% |
| 20 | 2898.622 | 94.883 | 96.73% |
| 21 | 1716.836 | 39.056 | 97.73% |
| AVERAGE | | | 95.96% |



Based on our gathered data for CPU vs GPU Runtimes for the Ray Tracer for different resolutions, we can conclude that running the program paralleling significantly improved the ray tracer's performance time. The CPU run time is incredibly long compared to the GPU's time due to the serialized instructions of iterating through each pixel and waiting for the color computation of each color. In contrast to this, we can see that with GPU, the time significantly improved due to each thread in the grid computing the color for its mapped image pixel. Based on our three results, we can see that as the resolution increases, the execution time of both the CPU and GPU increases; however, the CPU's time increase in execution time gets really long, which is not ideal for rendering an image for a 4k resolution monitor (nobody wants to wait 1 second + for their image to be rendered).

Overall, we see an improvement from using the CPU to GPU to render an image of the world. We reduced the execution time by 95%.

## Problems Faced:

**Environmental Setup:**

- Problem: The Ray Tracer program depends on methods defined in the unistd.h header file that is included in UNIX based environments, but is not natively supported on Windows.
  Solution: On Windows utilize MINGW compiler suite to take advantage of the compatibility headers allowing UNIX headers to run on Windows
- Problem: CUDA compilers (NVC/CC/C++) do not allow for specific C/C++ compilers to be designated and on Windows use the MSVC compiler and cannot be forced to use MINGW which removed UNIX Header Compatibility on Windows.
  Solution: Migrate project to Bender Server to take advantage of a preconfigured and shared UNIX-compatible environment.
- Problem: The Ray Tracer program requires C++ 17. Bender Server g++ version was outdated and did not support C++ 17.
  Solution: We use a singularity container that has C++ version that supports C++ 17.
  Note: The Bender Server C++ version eventually got updated, so we no longer need to use the Singularity container.
- Problem: The Ray Tracer program requires libpng library to produce images.
  Solution: Professor Wong was able to assist in getting libpng added to the singularity container, and once Bender was updated to support C++ 17 we stopped using singularity and libpng was already available on the main Bender environment

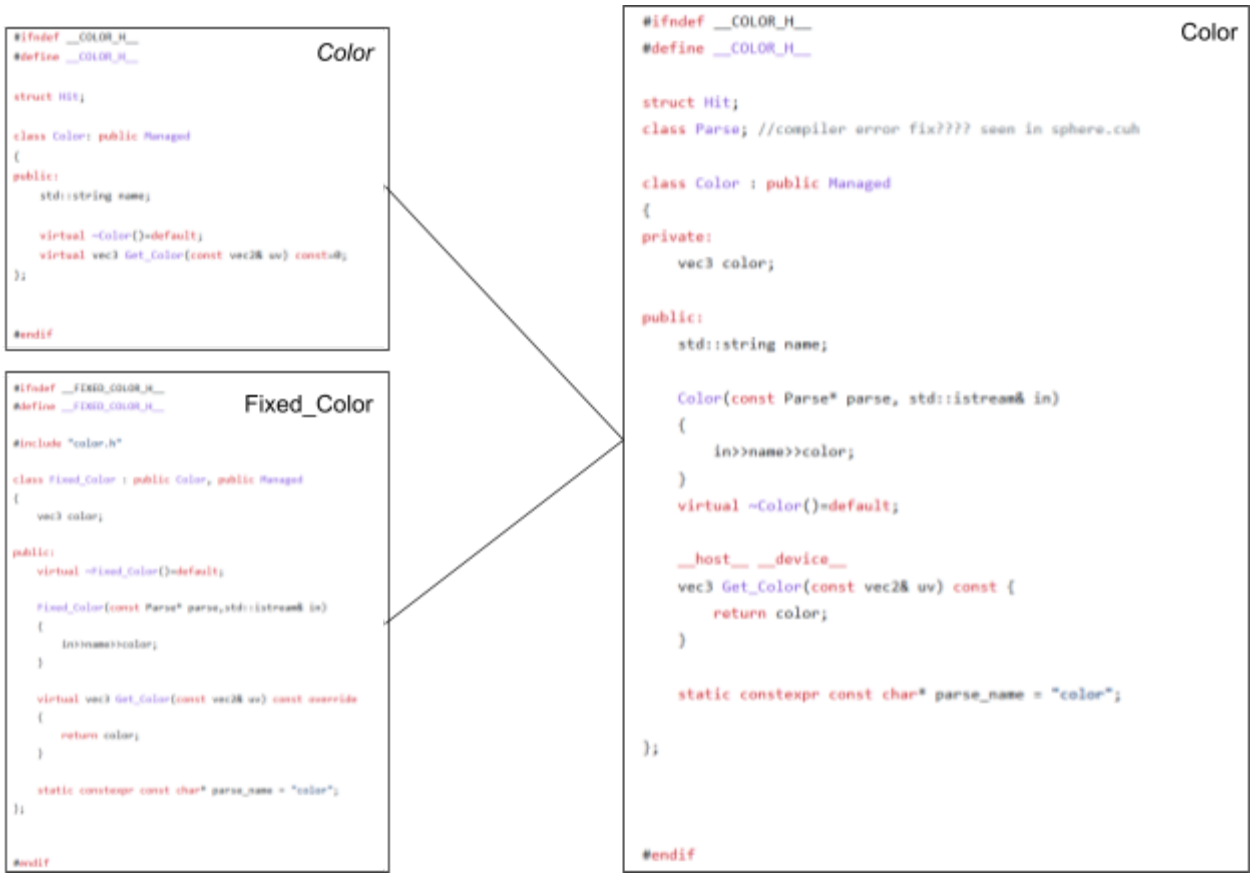
**Compilation Calls:**

- `nvc++ -std=c++17 -O3 *.cpp *.cu -o ray_tracer -lpng -lcudart -lcuda`
- Problem: Need to compile as C++ 17
  Solution: Replace NVCC calls with NVC++ and add the "-std=c++17" flag to ensure the correct C++ dialect was referenced.
- Problem: CUDA API calls in .cpp files were causing errors
  Solution: All .cpp files that depend on CUDA API calls had the line "#include<cuda_runtime.h>" added and the cuda libraries had to be linked within the compilation call so "-lcudart" and "-lcuda" were added.
- Problem: Some Files were converted from .h/.cpp to .cuh/.cu in order to better support CUDA programming
  Solution: Add "*.cu" to the call for all files to be compiled in addition to the "*.cpp" to ensure all CUDA and C++ source files in the directory were included in the program compilation.

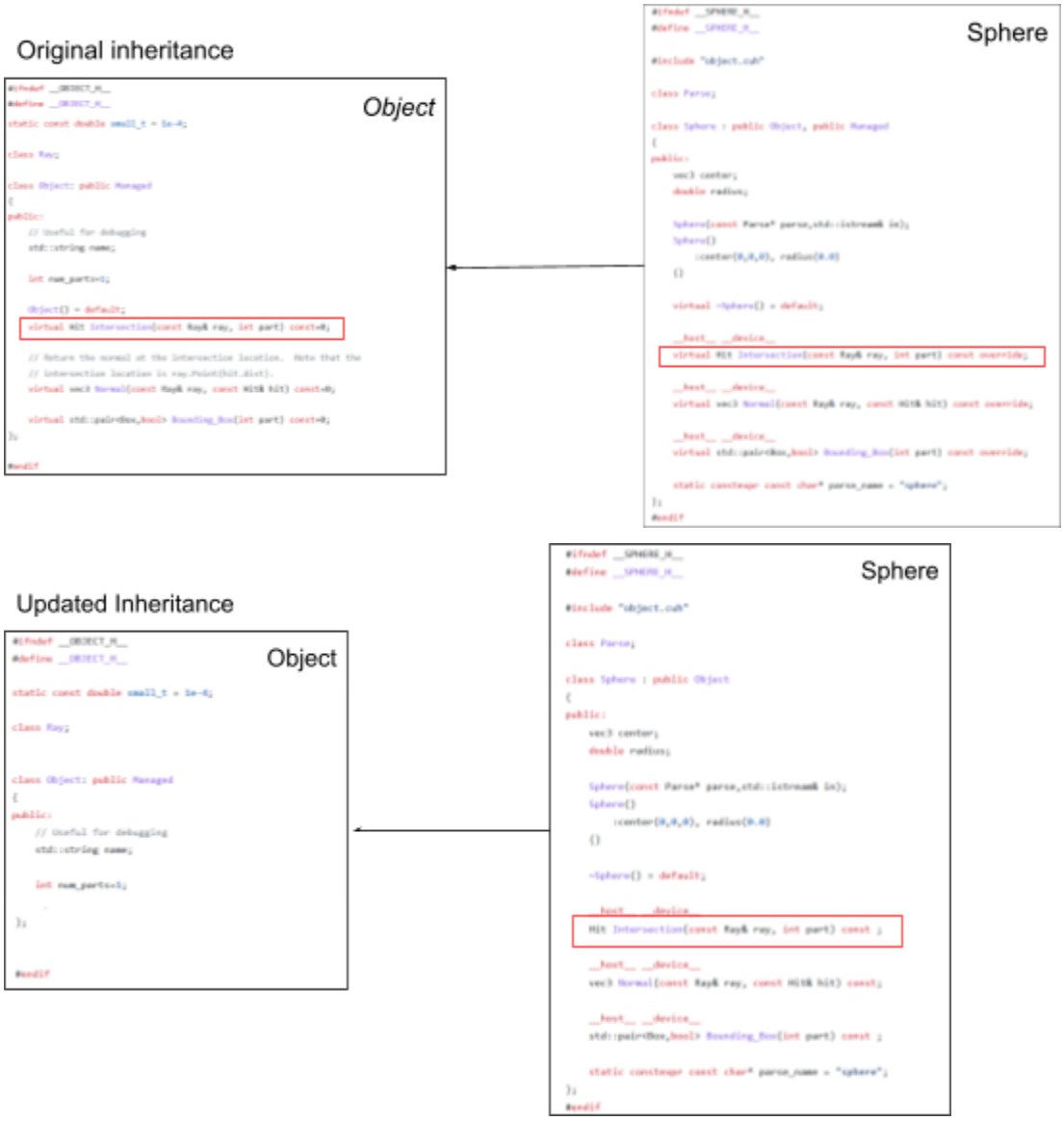
**Polymorphism:**

Problem: Abstract class objects cannot be created on the host side and passed to the device side:
Solution: Our final solution was to avoid polymorphism altogether. The following is how we handled it:

(1) Abstract classes that had single purposes were converted into a concrete class (for instance an abstract *Color* class with a Fixed_Color child was combined into a single concrete Color class)




(2) Abstract classes with multiple children had all pure virtual inheritance removed. Every child class was modified to have concrete definitions of all methods. Example below for the abstract object class with sphere child (plane child not shown):




(3) Structures dependent on the original abstract classes needed to be redefined and separated. Namely the `Shaded_Object` struct which, in the original code, holds an `Object*` and a `Shader*`. Since both Object and Shader classes were redefined (as both were abstract with multiple children), a new structure for each combination was made (see Figure 2).

(4) Modified parsing to create objects based on the new "hardcoded" struct combinations (input .txt files had to be changed to reflect this).

Original object creation

```
else if(token=="shaded_object")
{
    auto o=Get_Object(ss);
    auto s=Get_Shader(ss);
    render_world.objects.push_back({o,s});
}
```

Modified Object Creation (1/4)

```
else if (token == "flat_shaded_sphere"){
    auto o=Get_Sphere(ss);
    auto s = Get_Flat_Shader(ss);
    Flat_Shaded_Sphere fs;
    fs.sphere = o;
    fs.flat_shader = s;
    render_world.flat_shaded_spheres[render_world.num_flat_shaded_spheres++]=fs;
    //printf("Flat Shaded Spheres: %d", render_world.num_flat_shaded_spheres);
}
```

(5) Finally, any reference to obsolete classes and structs had to be fixed (namely `Shaded_Object`, `Object`, and `Shader`). For example anything that referenced `Shaded_Object` would have to be duplicated four times: one copy would reference a `flat_shaded_sphere`, another would reference a `phong_shaded_sphere`, another a `flat_shaded_plane`, and the last a `phong_shaded_plane`. The same goes for `Object` references (changed to reference `sphere` and `plane`) and `Shader` references (changed to reference `flat_shader` and `phong_shader`).

**Vector → Array:**
Problem: STL cannot be used inside kernels.
Solution: Convert the vectors used in our classes to arrays with a defined `ARRAY_SIZE`.

**Unified Memory:**
Problem: Since our program is written in C++, allocating memory, copying memory from host to device, and copying memory from device to host requires too much time to allocate and copy and will get too overwhelming and complicated with the number of objects that we would have to create and allocate on both the host and device side.
Solution: Using unified memory will allow for object creations to be easier because when we create new objects, memory is allocated in unified memory for both CPU and GPU to read and write to without having to allocate and copy between the CPU and GPU.

## Task Contribution Breakdown:

| Task Contribution Breakdown | |
|---|---|
| Environment Setup | **Tanisha: 0%**<br>**Justin: 100%** - set up github with ray tracer code, figured out all issues when running ray tracer, program dependencies, and required libraries, and found all the flags for CUDA compiling commands<br>**Vy: 0%** |
| Object Conversion to Unified Memory | **Tanisha: 25%** - converted 4 classes to unified memory, ran into issue with polymorphism<br>**Justin: 25%** - converted 3 classes, converted vector to array in render world and parse<br>**Vy: 50%** - researched unified memory, made example unified memory test programs for reference (reading and writing with unified memory in host and kernel), and made class member functions available to host and device, converted 5 classes to unified memory, found the cause of incompatible polymorphism with kernel launch |
| Parse and Object Creation | **Tanisha: 50%** - modified parse code to find the mapped object and create the correct object for the intended structure and add the object to the correct object grouping array in render world, modified computation code to be compatible with the new non-polymorphism design, and tested that the objects are added to the correct array in render world<br>**Justin: 15%** - debugged the computation code to be compatible with the new structure of no polymorphism, fixed the issue with phong shader<br>**Vy: 35%** - changed all abstract classes to normal classes, created structures for the shaded object combination, created arrays to store these combinations, create mapping for object creations in parse |
| Kernel Design | **Tanisha: 0%**<br>**Justin: 100%** - calculated the grid size and mapped each thread in the grid<br>to the output pixel with boundary conditions to |

| | |
|---|---|
| | be within image size<br>**Vy: 0%** |
| Data Gathering | **Tanisha: 0%**<br>**Justin: 35%** - changed test .txt files for both cpu and gpu, wrote testing script to test all cpu and gpu tests<br>**Vy: 65%** - recorded cpu and gpu runtimes for three different resolutions, graphed the times of each test # for the three resolutions, analyzed and compared the cpu and gpu performance |
| Report Drafting | **Tanisha: 30%**<br>**Justin: 30%**<br>**Vy: 40%** |
| Presentation Slides | **Tanisha: 60%**<br>**Justin: 20%**<br>**Vy: 20%** |