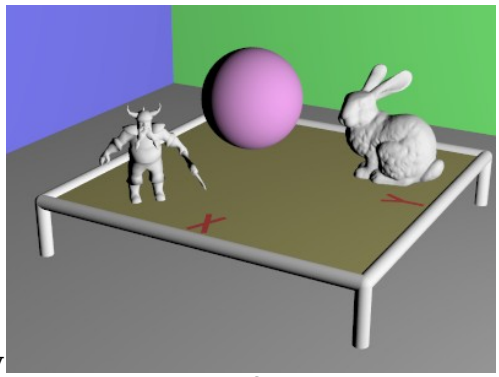
CS500 Project 1

Synopsis

Implement ray-casting and a spatial data structure as a first step toward a full featured ray-tracer. Start with **Project 0** and the distributed framework for code and instructions on parsing the scene and writing out the computed image.



Ray-cast output of test scene. Your shapes should match exactly, but lighting may differ.

Instructions

Your ray-tracer executable should accept a scene filename (as command line argument), read and parse the scene described in the file, build an internal representation of the scene, ray-cast it, and output the final image into an HDR image file. For efficiency sake, the internal representation should be in a spatial data structure to accelerate the ray/object calculations.

Ray-casting

The ray-casting task involves the following operations:

- For each pixel on the screen, generate a ray, from the eye through the pixel. Make sure to use OpenMP to parallelize the main Y loop.
- Find the ray's front-most intersection point with the scene's objects.
- Calculate a color for the intersection point. Make this as simple as you wish, perhaps just a diffuse reflection calculation: $(N \cdot L) K_d$ for an appropriate vector L toward the light. We'll do a proper lighting in a later project.
- Write that color to the pixel for which the ray was generated.

The ray-tracer should handle objects of type "**sphere**", "**box**", "**cylinder**", and "**triangle**". No reflections, refractions, or other typical ray tracing effects should be implemented yet.

Spatial data structure (Details on last page)

The test scene contains nearly 70,000 objects, mostly triangles from the Stanford bunny. Your first pass should test every ray (one per pixel) against every object. This is expected to take about a minute on a modern CPU. With the application of a spatial data structure, this can be sped up by a factor of several hundred.

The framework provides a library called *"A modern C++ BVH construction and traversal library"* found at **https://github.com/madmann91/bvh** .

Each object in your **std::vector<Shape*>** list of objects will need a bounding box. The **SimpleBox** class (derived form bvh::BoundingBox) inherits these useful methods for constructing bounding boxes:

```
SimpleBox()                  // Empty box
SimpleBox(vec3 point)        // Box surrounding a single point
SimpleBox.extend(vec3 point) // Grows a box to include the new point.
```

Programming hints

Vectors: The framework provides GLM, a vector/matrix/quaternion library. See **geom.h** and **geom.cpp** for a list of objects and procedures.

Dynamic memory: Use all the dynamic memory (news and deletes) you want during the reading of the scene and creation of the **std::vector<Shape*>** list of objects, and the BVH, but for efficiency sake, **DO NOT** allocate dynamic memory during the body of the ray-tracing application**.**

For debugging purposes, consider:

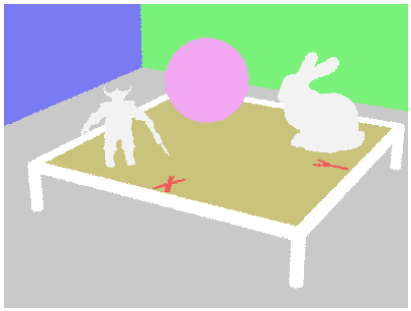
An acceleration structure that intersects a ray with every object on the list sequentially.
An acceleration structure that intersects a ray with every object's bounding box.

What to submit

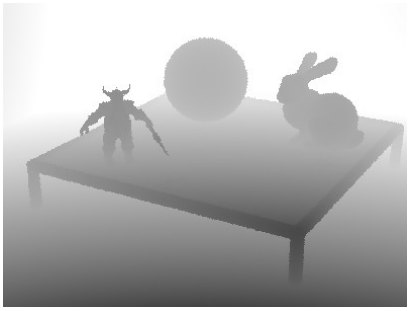
Turn in a zip file containing a **project report** and all your **source code**.

The report should outline the project and describes how you implemented, exercised, tested and verified it. The report will naturally contain images generated by your ray-tracer, but **more important** is the text that describes the images and why they are relevant. The more you say about your project, the better, but it should include at least the following:

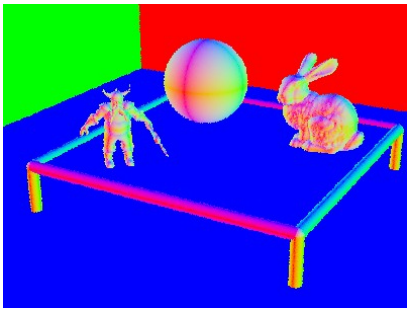
- Verify that your intersection calculations are correct by drawing something unique to each object – perhaps the material's Kd.
- Verify that each ray's intersection t value is calculated correctly by drawing it as a gray-scale adjusted to $[0,1]$ range. For the test image, a gray-scale value of $(t-5)/4$ works well.
- Verify that the normal of each intersection point is calculated correctly by drawing the normal (or its absolute value) at each point.
- Verify that the spatial data structure provides the expected speedup. There will be no speedup if the scene is too simple, for instance if the Stanford bunny is commented out of the provided test scene.



Output of objects base color. No lighting calculation whatsoever.



Output of the t value of the ray's intersection, adjusted for display purposes to [0,1].



Output of (the component-wise absolute value) of the surface normal.

The source code should contain only **.cpp** and **.h** files, and perhaps project and solution files. It should not contain compiler generated output, including the Debug and Release folders, and certainly not the **.vs** folder (which can expand into several gigabytes, none of which are useful to me).

An acceleration data structure

The framework provides a library for accelerating the intersection of a ray with the full scene. It should provide a huge speedup compared to the default of intersecting a ray with every object on the scene's list of objects. (Well over a factor of 200 on the test scene containing the Stanford bunny.)

Reference: See *"A modern C++ BVH construction and traversal library"* at **https://github.com/madmann91/bvh**
Most of the sample code comes from their example at
https://github.com/madmann91/bvh/blob/master/test/custom_primitive.cpp

The sample code (distributed in file **acceleration.h** and **acceleration.cpp**) provides:

- **class AccelerationBvh** to encapsulate the building and use of the structure
  - The constructor **AccelerationBvh(std::vector<Shape*>)** builds and stores the BVH
  - The method **Intersection intersect(const Ray&)** intersects a ray with the scene and returns the ray's first intersection. The returned **Intersection** record must be sufficiently general to record an intersection with any object, or with no objects.
- **class BvhShape** meant to wrap a **Shape*** instance and provides:
  - A specification of two types for the bvh template machinery:
    using ScalarType = float
    using IntersectionType = Intersection
  - Method **bounding_box()** returns a shape's bounding box.
  - Method **center()** returns that bounding box's center point.
  - Method **intersect(const bvh::Ray<float>& bvhray)** intersects a ray with the shape and returns in **Intersection** record or a **std::nullopt.**
- **class SimpleBox** which extends the **bvh::BoundingBox** with methods that take **glm::vec3** instead of the original **bvh::Vector3**.
- Conversions between my ray type **Ray**, and the bvh ray type **bvh::Ray**
- Conversions between **glm::vec3** and **bvh::Vector3**