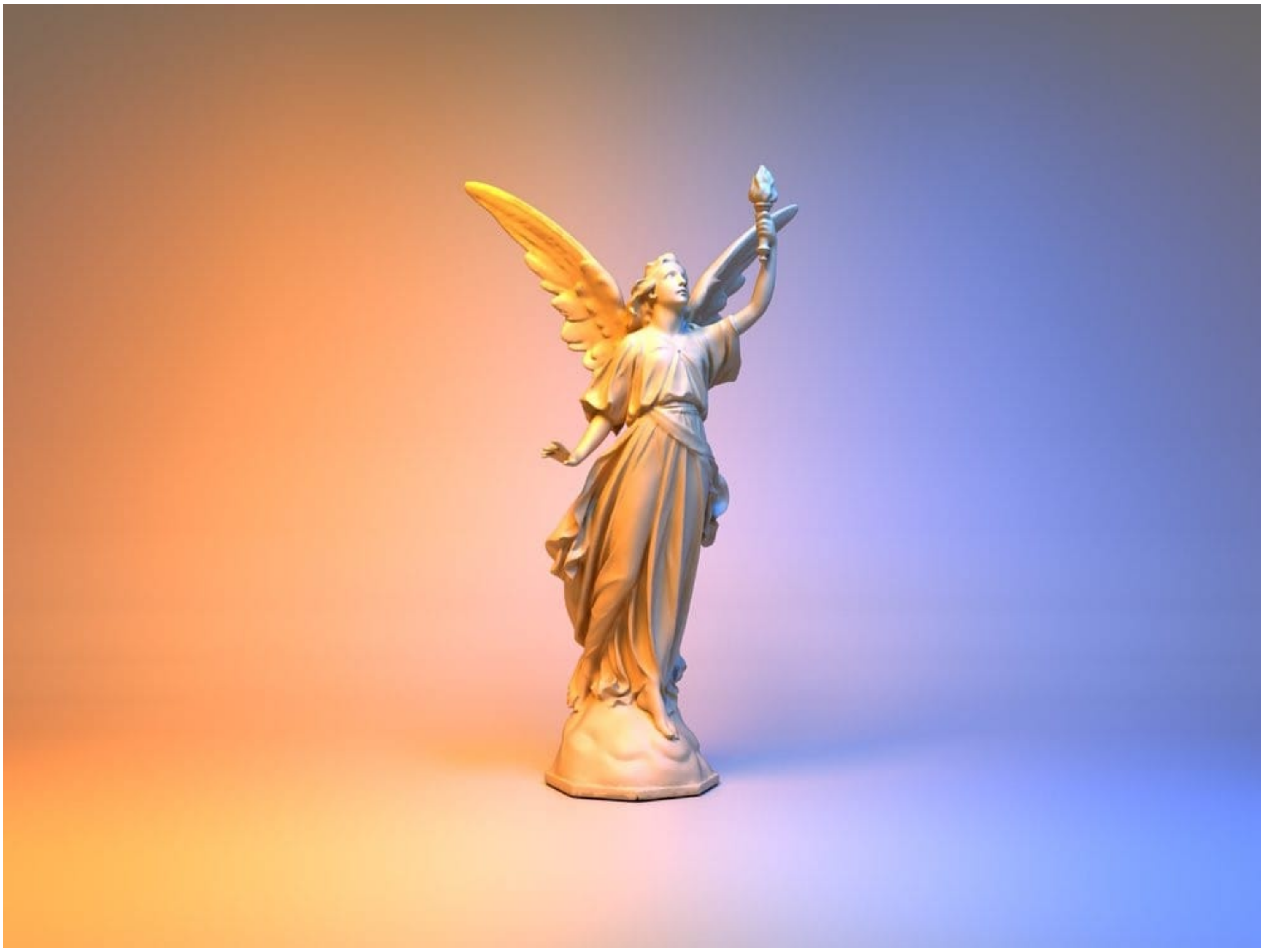


Assignment 2: Ray Tracing

[Deadline: Monday, 14th Oct 2019]

Flag and parameters	Description
`-s <INT>`	Number of camera rays per pixel (default=1, should be a power of 2)
`-l <INT>`	Number of samples per area light (default=1)
`-t <INT>`	Number of render threads (default=1)
`-m <INT>`	Maximum ray depth (default=1)
`-f <FILENAME>`	Image (.png) file to save output to in windowless mode
	Width and height in pixels of output image (if windowless) or of

`-r <INT> <INT>`	GUI window
`-p <x> <y> <dx> <dy>`	Used with the -f flag to render a cell
`-c <FILENAME>`	Load camera settings file (mainly to set camera position when windowless)
`-a <INT> <FLOAT>`	Samples per batch and tolerance for adaptive sampling
`-H`	Enable hemisphere sampling for direct lighting
`-h`	Print command line help message

Moving the camera (in edit and BVH mode)

Command	Action
`Rotate`	Left-click and drag
`Translate`	Right-click and drag
`Zoom in and out`	Scroll
`Reset view`	Spacebar

Keyboard commands

Command	Keys
`Mesh-edit mode (default)`	E
`BVH visualizer mode`	V
`Descend to left/right child (BVH viz)`	LEFT/RIGHT
`Move up to parent node (BVH viz)`	UP
`Start rendering`	R
`Save a screenshot`	S
`Decrease/increase area light samples`	- +
`Decrease/increase camera rays per pixel`	[ ]
`Decrease/increase maximum ray depth`	< >

`Toggle cell render mode`	C
`Toggle uniform hemisphere sampling`	H
`Dump camera settings to file`	D

Cell render mode lets you use your mouse to highlight a region of interest so that you can see quick results in that area when fiddling with per pixel ray count, per light ray count, or ray depth.

Ray Generation and Scene Intersection

Task 1: Filling in the sample loop

Fill in PathTracer::raytrace_pixel() in pathtracer.cpp. This function returns a Spectrum corresponding to the integral of the irradiance over this pixel, which you will estimate by averaging over ns_aa samples.

The inputs to this function are integer coordinates in pixel space. You should generate ns_aa random rays through this pixel using camera->generate_ray() (which you will implement in Task 2) and evaluate their radiance with trace_ray().

Notes:

- PathTracer owns a gridSampler, which has a method you can use to get random samples in $[0,1]^2$ (see sampler.h/cpp for details).
- The width and height of the pixel buffer are stored in sampleBuffer.w and sampleBuffer.h.
- You will most likely want to pass a location to the camera that has been scaled down to $[0,1]^2$ coordinates.
- When ns_aa == 1, you should generate your ray through the center of the pixel, i.e., $(x+.5,y+.5)$.
- Remember to be careful about mixing int and double types here, since the input variables have integer types.

Task 2: Generating camera rays

Fill in Camera::generate_ray() in camera.cpp. The input is a 2D point you calculated in Task 1. Generate the corresponding world space ray as depicted in this slide.

The camera has its own coordinate system. In camera space, the camera is positioned at the origin, looks along the $-z$ axis, has the $+y$ axis as image space "up". Given the two field of view angles hFov and vFov, we can define a sensor plane one unit along the view direction with its bottom left and top right corners at

```
  Vector3D(-tan(radians(hFov)*.5), -tan(radians(vFov)*.5),-1)
  Vector3D( tan(radians(hFov)*.5),  tan(radians(vFov)*.5),-1)
```

respectively. Convert the input point to a point on this sensor so that $(0,0)$ maps to the bottom left and $(1,1)$ maps to the top right. This is your ray's direction in camera space. We can convert this vector to world space by applying the transform c2w. This vector becomes our r.d (don't forget to normalize it!!). The r.o parameter is simply the camera's position pos. The ray's minimum and maximum $t$ values should be the nClip and fClip camera parameters. ***Tasks 1 and 2 will be tricky to debug before implementing part 3, since nothing will show up on your screen!***

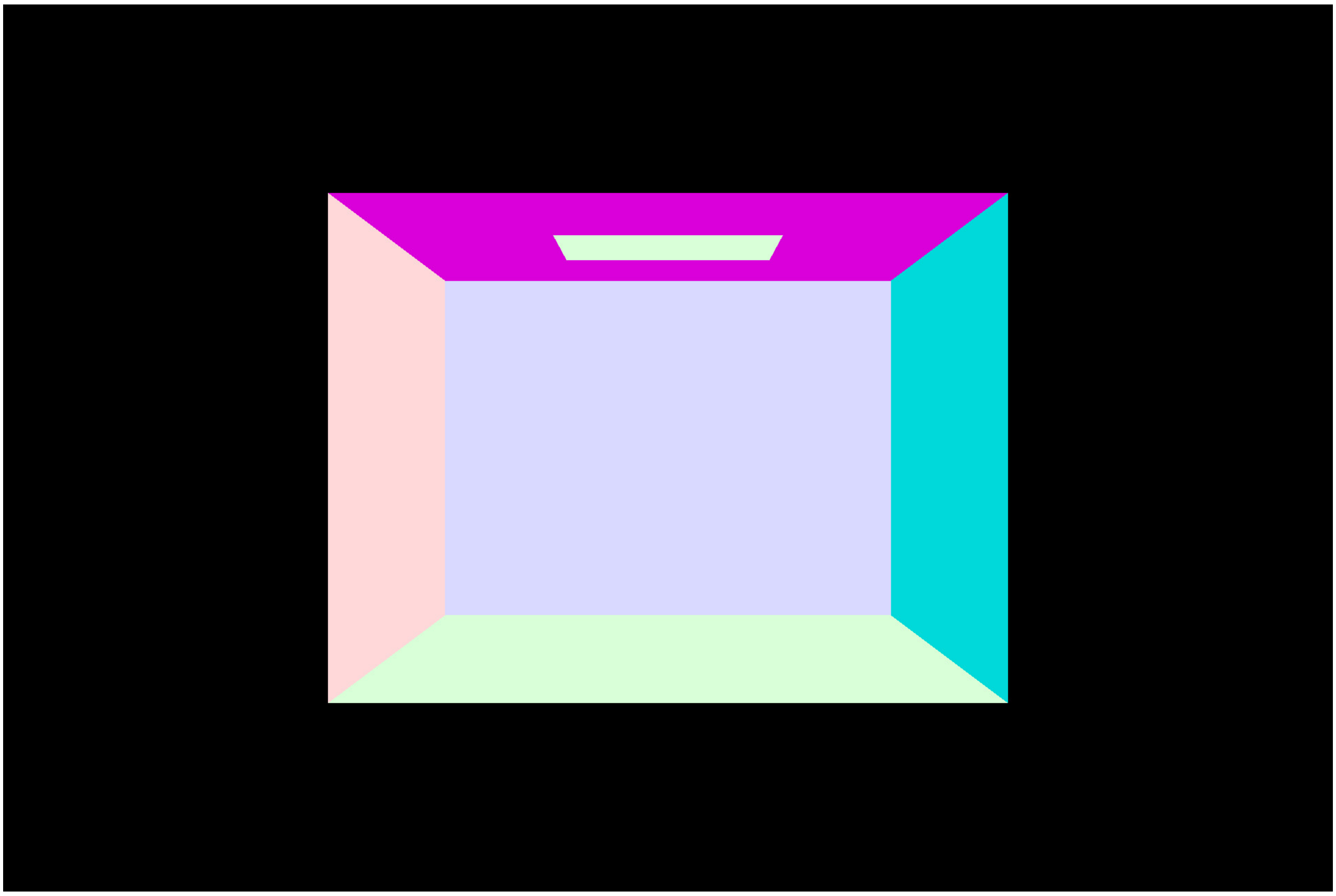
Task 3: Intersecting Triangles

Fill in both Triangle::intersect() methods in triangle.cpp. You are free to use any method you choose, but we recommend using the Moller Trumbore algorithm. Make sure you understand the derivation of the algorithm: here is one reference.

Remember that not every intersection is valid -- the ray has min_t and max_t fields defining the valid range of t values. If t lies outside this range, you should return false. Else, update max_t to be equal to t so that future intersections with farther away primitives will be discarded.

Once you get the ray's $t$-value at the intersection point, you should populate the Intersection *isect structure in the second version of the function as follows:

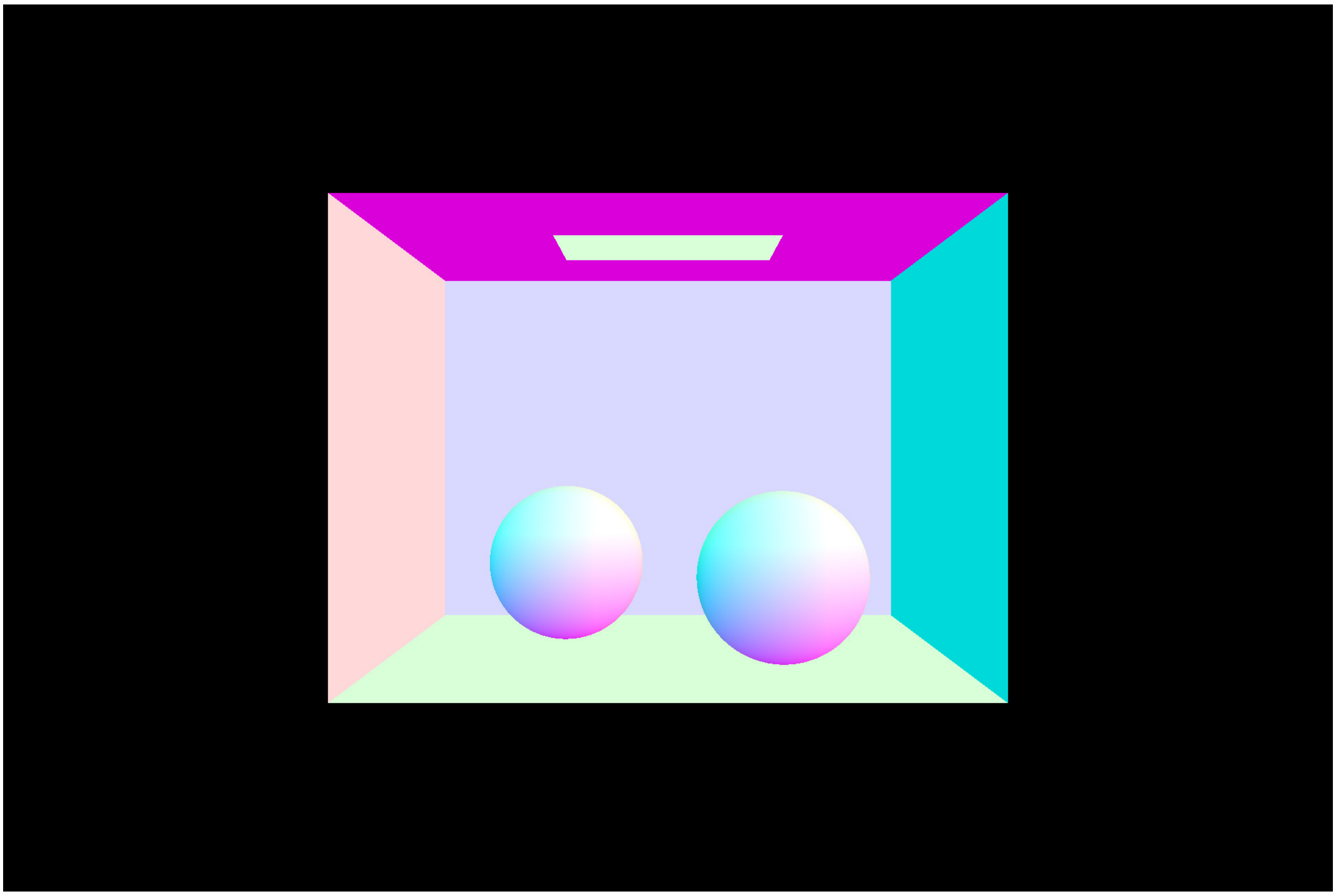
- `t` is the ray's t$t$-value at the hit point.
- `n` is the surface normal at the hit point. Use barycentric coordinates to interpolate between `n1, n2, n3`, the per-vertex mesh normals.
- `primitive` points to the primitive that was hit (use the `this` pointer).
- `bsdf` points to the surface bsdf at the hit point (use `get_bsdf()`).



Task 4: Intersecting Spheres

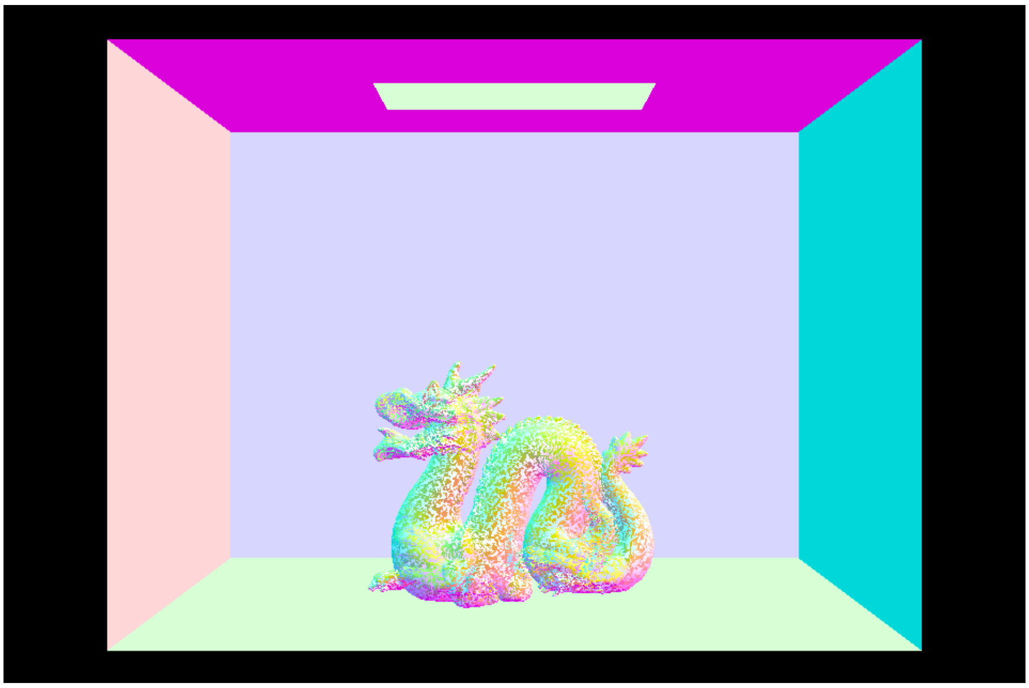
Fill in both Sphere::intersect() methods in sphere.cpp. Use the quadratic formula. There is also a helper function Sphere::test() that you might want to implement and use. As with Triangle::intersect(), set r.max_t in both routines and fill in the isect parameters for the second version of the function. For a sphere, the surface normal is a scaled version of the vector pointing from the sphere's center to the hit point.

Now the spheres should appear in dae/sky/CBspheres_lambertian.dae:



Reference: Berkeley course - https://cs184.org/article/12

Additional info



This result was generated using the following command and took 2194.49 seconds to render.

./pathtracer -t 8 -r 480 320 ../dae/sky/CBdragon.dae

Bounding Volume Hierarchy

Task 5: Constructing the BVH

Implement the function `BVHAccel:construct_bvh()` inside *bvh.cpp*. The `BVHAccel` class itself only contains a `BVHNode *root`. Each node contains a bounding box `bb`, left and right children `l` and `r`, and a pointer `vector<Primitive *> *prims` to a list of actual scene primitives. For interior nodes, `l` and `r` are non-`NULL`, and for leaf nodes, `prim` is non-`NULL`.

The starter code creates a one-node BVH by storing all nodes directly into a leaf node. You may notice that any *.dae* files with even mildly complicated geometry take a very long time to "render," even with only the simple normal shading. Even a BVH constructed with simple heuristics will perform much better (taking ray intersection complexity from linear to log, for those who care).

Some important utility functions for you to note:

1. `Primitive::get_bbox()` returns the bounding box of a primitive.
2. `BBox::expand()` expands a bounding box to include the function argument, which can either be a `Vector3D` or another `BBox`.
3. Check out the `Vector3D` members variables inside a `BBox`: `min, max, extent`, where `extent = max-min`.

We recommend that you first attempt BVH construction with some the following simple (but slightly inefficient) recursive function:

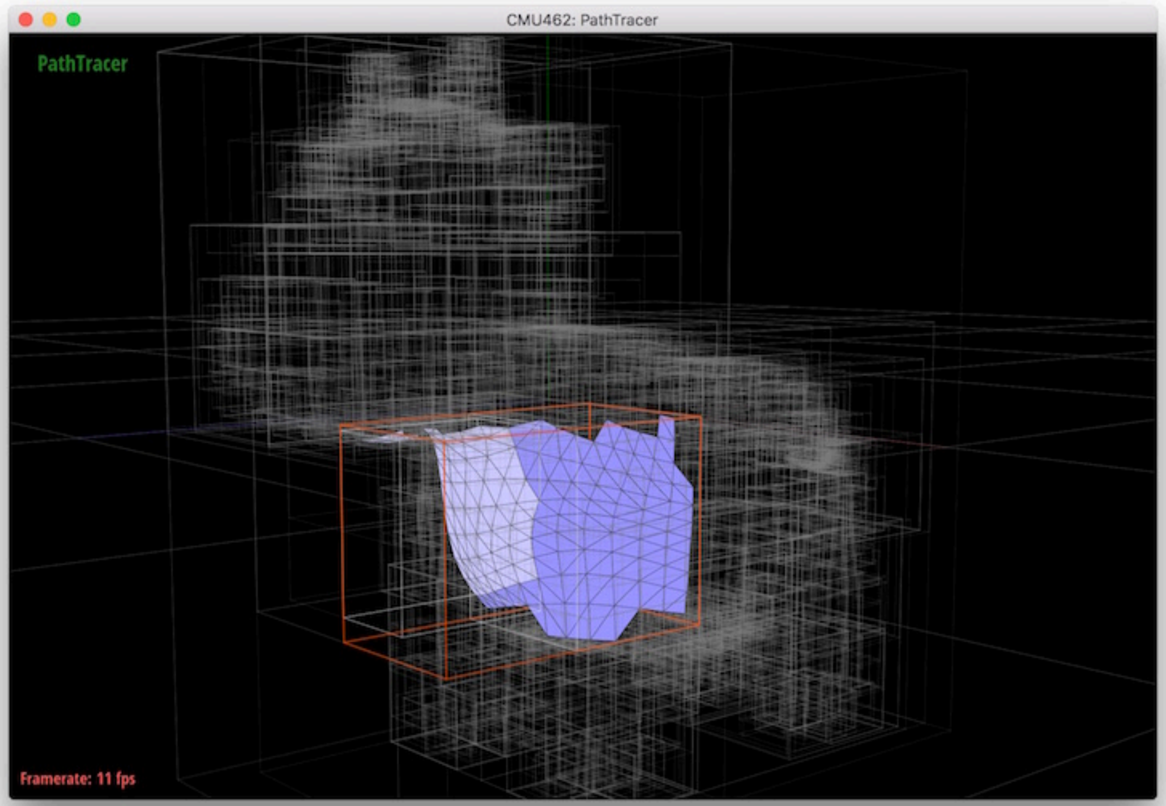
1. Compute the bounding box of the primitives in `prims` in a loop.
2. Initialize a new `BVHNode` with that bounding box.
3. If there are at most `max_leaf_size` primitives in the list, this is a leaf node. Allocate a new `Vector<Primitive *>` for node's primitive list (initialize this with `prims`) and return the node.
4. If not, we need to recurse left and right. Pick the axis to recurse on (perhaps the largest dimension of the bounding box's extent).
5. Calculate the split point you are using on this axis (perhaps the midpoint of the bounding box).
6. Split all primitives in `prims` into two new vectors based on whether their bounding box's

centroid's coordinate in the chosen axis is less than or greater than the split point. (`p->get_bbox().centroid()` is a quick way to get a bounding box centroid for `Primitive *p` .)

7. Recurse, assigning the left and right children of this node to be two new calls to `construct_bvh()` with the two primitive lists you just generated.
8. Return the node.

~~One potential problem to consider: what happens if all the primitives lie on one side of the split point? We will get a segfault because of infinite attempted recursive calls. You need some logic to handle the case where either the left or right primitive vector is empty.~~

We've provided a helpful BVH visualization mode which you can use to debug your implementation. To enter this mode, press `V` . You can then navigate around the BVH levels using the left, right, and up keys. The BVH viz view looks like this:



~~Task 6: Intersecting `BBox`~~

~~Implement the function `BBox::intersect()` inside *bbox.cpp*, using the simple ray-aligned-plane intersection equation here and the ray-aligned-box intersection method here. Note that this function returns an *interval* of `t` values for which the ray lies inside the box.~~

Task 7: Intersecting `BVHAccel`

Using the previous two parts, implement the two `BVHAccel::intersect()` routines inside *bvh.cpp*. The starter code assumes that *root* is a leaf node and tests the ray against every single primitive in the tree. Your improved method should implement this recursive traversal algorithm.

Notes

- You can safely return `false` if a ray intersects with a `BBox` but its `t` interval has an empty intersection with the ray's interval from `min_t` to `max_t`.
- In the version with no `isect` parameter, you can safely return `true` after a single hit. However, the other version must return the *closest* hit along the ray, so it needs to check every `BBox` touched by the ray.
- If all primitives update `r.max_t` correctly in their own intersection routines, you don't need to worry about making sure the ray stores the closer hit in `BVHAccel::intersect()` since it will be taken care of automatically.

Once this part is complete, your intersection routines should be fast enough to render any of our scene files in a matter of seconds (with normal shading only), even ones like *dae/meshedit/maxplanck.dae* with tens of thousands of triangles:

Or *dae/sky/CBlucy.dae*, with hundreds of thousands of triangles: