Assignment 2: Ray (Computer Graphics)

Introduction

In this project, you'll implement a pretty fully-featured raytracer. Raytracing is a really elegant algorithm that allows for a lot of physically-realistic effects with a relatively small amount of code (Paul Heckbert fit a raytracer on the back of his business card!)

Complete this project in groups of 2. Instructions for group setup are below.

There are 11 separate `TODO`s, some of which have multiple parts. Most of the TODOs involve less than 10 lines of code, though some will be more. I ran `sloc` on my solution and the skeleton and the difference in lines of actual code ("source") was less than 200. However, many of those lines are backed by math that you need to understand to be able to write them.

Getting Started

Setting up the project

On any computer where you intend to work on this project, you'll need to complete the following steps to get the project environment set up in Julia. Start by cloning the repository, enter the RayA2 directory, and fire up Julia:

```
$ cd path/to/RayA2
$ julia --project
```

The `--project` flag tells Julia to look in the current directory for a Project.toml file; when found, this automatically activates the project's environment. At the Julia prompt, press `]` to switch to the `pkg` prompt:

```
julia> ]
(RayA2) pkg>
```

The `instantiate` command will now install the package dependencies for this project:

```
(RayA2) pkg> instantiate
Cloning default registries ...
```

Now press backspace to get out of the `pkg` prompt back to the `julia` prompt. We'll now make the code in the `RayA2` module accessible for use at the REPL:

```
(RayA2) pkg> [backspace]
julia> using RayA2
```

```
[ Info: Precompiling RayA2 (...) - this might take a moment
julia>
```

At this point, the project should be all set up. To verify that it is, run the following:

```
julia> RayA2.main(1, 1, 300, 300, "../results/out_0.png")
```

and verify that `out_0.png` has been created in the `results` directory; it should be a 300x300 image containing all black pixels. This is your starting point - your job in this project is to make the picture more colorful!

Running from the REPL

Running a new instance of Julia every time we want to run our code, as we did in A1, is slow. This is because Julia spends a lot of time on startup just loading and precompiling modules and your code. The best way to avoid this is to develop your code from the REPL instead of from the shell. That is, start up an instance of `julia`, then make a call to a `main` function or somesuch every time you wish to run your code.
Unfortunatley, Julia does not have the built-in ability to notice that your code has changed and run the latest version. Fortunately, there is a package that called `Revise.jl` that helps with this. To use it, simply run `using Revise` at the Julia prompt before running `using RayA2`. Then, you can make repeated calls to `RayA2.main` (or other functions if you'd like) and your changes will be noticed. It's important to make sure that you run `using Revise` *before* running `using RayA2`.

Here's what you should do each time you want to get started working on the project:

```
$ cd path/to/RayA2
$ julia --project
julia> using Revise
julia> using RayA2
[Info: Precompiling RayA2 [...]
# now you can call your code at will:
julia> RayA2.main(...)
```

The `--project` flag works in place of running `]activate .` after starting julia; either one works. If you wish to set Julia up to use Revise every time you start by default, see the instructions here.

Julia Concepts

The Julia language is different from languages you're probably used to in a variety of ways, and some of them are important to understand for this project. This section provides an overview of some of these concepts.

Multidimensional Arrays

The familiar `Vec3` and `Vec2` types you used in A1 are special examples of Julia's natively supported Multidimensional Arrays. Another example is the `canvas` in `RayA2`'s main function, for example, is a `height`-by-`width` 2D array of a special color type (`RGB{Float32}`, which represents a color in RGB format with 3 single-precision floats). Arrays such as `canvas` and `Vec3`s (for which I used Static Arrays, because their length is known and fixed) support a bunch of really nice functionality that most other languages don't have built in:

- Indexing: you can access values from `canvas` as `canvas[i,j]` to get the pixel at the `i`th row and `j`th column (both starting at (1,1) in the top left) of the image. The first element of a `Vec3` called `x` would be accessed using `x[1]`. See here for full details.
- Elementwise arithmetic: two arrays of the same shape can be added together, resuliting in a new array of the same shape. See here for more details.
- Iteration: you can loop over elements of an array (`for val in A`), or over each of its indices (`for i in eachindex(A)`). The eachindex thing even works for multi-dimensional arrays! Details here.

Types

Types in Julia are optional: you *can* tell Julia what type something is, but you don't *have* to. The general reasons to specify types are as assertions (to make catching bugs easier) and performance (to give the compiler more information). The consensus at this point, though it's still early, seems to be that you should only assert types if you have a good reason to. I'm not sure all the skeleton code follows this advice very well, but I tried.

Type annotations look like this:

```
3::Int
```

or

```
3.0::Float64
```

The most important time to pay attention to types in this project is in function signatures where the arguments of a function are typed. For example, in `Cameras.jl` we have a the function header:

```
function pixel_to_ray(camera::CanonicalCamera, i, j)
```

The first argument must be of type `CanonicalCamera`, whereas the second and third can be anything (formally, they are of type `Any`, which is a supertype of all types). From context, they should probably be integers, but I didn't bother to tell the compiler that. These type annotations play a major role in the next section.

See https://docs.julialang.org/en/v1/manual/types/ if you wish to geek out over the full details of Julia's type system.

Multiple Dispatch

The biggest departure from most languages you've probably encountered is that Julia is opinionatedly **not** object-oriented. Instead of object-oriented programming, class hierarchies, polymorphism, etc. etc. etc., Julia uses a paradigm called **multiple dispatch** to enable modular design. All the software architecture decisions in the raytracer have more or less been made for you, but nonetheless it's important for you to understand how the raytracer is designed.

The basic idea of multiple dispatch is that each **function** (i.e., a subroutine with a particular name) can have multiple **methods**, each of which handles different numbers of arguments and types. You can think of this as method overloading on steroids. This ends up being a powerful and flexible replacement for many object-oriented patterns and constructs.

Let's look at an example of this in the skeleton code. In `RayA2.jl`, there is a function called `determine_color`. Its purpose is to determine the color seen along a given ray - basically, this is most of the inner loop of the raytracer. The function takes several arguments that are needed to do the shading calcuation, but in particular notice its first parameter, `shader`. There are two separate definitions of the function, one with `shader::Flat`, and another with `shader::Normal`. At runtime, when we call `determine_color`, the type of the `shader` argument will

determine which of these methods is called. One of your tasks will be to implement a third **method** of this **function**.
In an object-oriented language we might have a Shader superclass with an abstract `determine_color` method implemented by each of its subclasses. Multiple dispatch allows you decide which implementation of a function is called based on any combination of its arguments' types. In this project, we keep it mostly pretty simple - most functions dispatch on only one of their arguments' types.

Packages, Modules, Submodules, `import`, `using`
Once again the details here are mostly taken care of, but you'll notice this project is structured a bit differently from A1, in that all the modules in separate files are **submodules** of the `RayA2` module. They are included using `include` statements, which work like you'd expect. Even after including the files, the code in `RayA2` still doesn't have automatic access to their functionality because it is wrapped in submodules. There are two ways to get access to names in other modules:

- `import MyModule` provides access to qualified names in `MyModule`, as in `MyModule.some_function()`
- `using MyModule` exposes all names in `MyModule` that are explicitly `export`ed. It also has the same effect as `import` such that all names, regardless of whether they are `export`ed, are accessible when qualified with the module name.

Tasks - Overview

In this assignment, you will complete the following tasks:

1. Intersect a ray with a sphere
2. Find the closest intersecting object along a ray
3. Implement the main raytracing loop that determines the color of each pixel.
4. Implement diffuse (Lambertian) and diffuse+specular (Blinn-Phong) shading.
5. Add shadow support
6. Add support for mirror-reflective materials
7. Implement ray-triangle intersection so we can render triangle meshes.
8. View ray generation: Implement a general perspective camera
9. Add support for textures
10. Model your own scene and render a cool image!
11. 580 only: Complete at least one extension

Tasks - Detail

0. Barebones ray generation (completed for you)

Basic ray generation for a canonical perspective camera has been implemented for you. See the `CanonicalCamera` type and the `pixel_to_ray` function in `Cameras.jl`. This camera is at the origin looking down the -*z* axis with a viewport width of 1 and a focal length of 1, for a horizontal field of view of 60 degrees.

1. Ray-sphere intersection

The next thing we need is to tell whether a ray intersects an object. We start with spheres. Your task is to implement `ray_intersect(ray::Ray, object::Sphere` in `Scenes.jl`. If the ray does not intersect the sphere, this method should return `nothing`; otherwise, it should return a `HitRecord` struct containing information about the intersection. The fields of the `HitRecord` should be filled in as follows:

- `t` is the value of the ray's parameter *t* at which the ray intersects the sphere. If there are two intersections, use the one with smaller `t`.
- `intersection` is a Vec3 giving the position of the intersection point
- `normal` is a Vec3 giving the normal vector of the sphere's surface at the intersection point
- `uv` is the texture coordinate or `nothing`; we'll come back to this later, so for now set this to `nothing`
- `object` is the `Sphere` object that was hit by the ray

2. Find the closest intersection

With multiple objects in the scene, we need to make sure that we base the pixel's color on the closest object. Fill in `closest_intersect` in `ray.jl`. This function takes a `Scene`, a `Ray`, and bounding values `tmin` and `tmax`. We'll use `tmin` and `tmax` for a few different purposes, but most obviously we need to make sure we don't get ray intersections behind the camera (i.e., where `t` is less than 1, or whatever the focal distance of the camera is). The `closest_intersect` function should call the `ray_intersect` on each object in the `scene` and return the resulting `HitRecord` object for the closest intersection whose `t` value lies between `tmin` and `tmax`. If no objects are intersected, it should return `nothing`.

3. Write the main raytracing loop

Take a look at the `traceray` function. This currently performs two tasks:

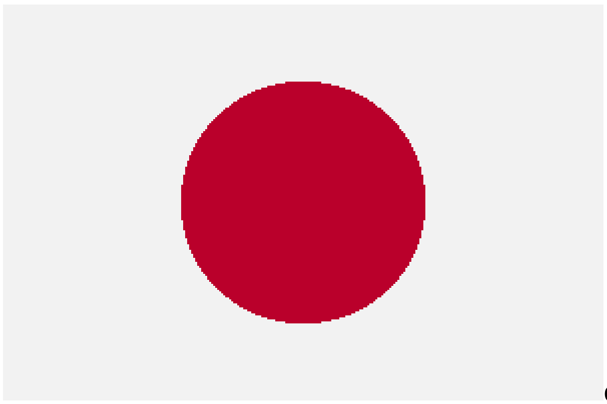
- It calls `closest_intersect` to get a HitRecord for the closest object along the ray; If there is no object, it returns the background color associated with the scene.
- It calls `determine_color` to choose the color for the pixel based on the information in the `HitRecord`. This will already work correctly for objects with the `Flat` shading model, because `determine_color` simply returns the diffuse color for such objects.

You're finally about to have a working raytracer! Fill in the missing part of the `main` function in `ray.jl`. For each pixel in the image, call `Cameras.pixel_to_ray` to generate a viewing ray for that pixel. Then, call the `traceray` function and set the resulting color to that pixel of the canvas. Because we only want to consider objects in front of the camera, set `tmin` to 1 and `tmax` to `Inf`. For now, `traceray` ignores the `rec_depth` parameter; you can leave it out and it will default to 1.

At this point, you should be able to run

```
julia> RayA2.main(1, 1, 200, 300, "results/out_1.png")
```

and see an image like this:


out1

Whoa! We just rendered something like the Japanese flag!

4. Shading - diffuse and diffuse+specular

That’s pretty cool, but doesn’t look like a physically realistic scene - everything is flat-colored. To get towards more realistic-looking scenes, we need **lights** and **shading**. The file `Lights.jl` contains types for lights that can be included in a scene. For this assignment, we’ll support **directional** lights and **point** lights.

4a. Calculating Light Directions

The key thing we’ll need to know when deciding a pixel’s color due to a given light is the direction of the light from the pixel. In `Lights.jl`, implement the two methods of the `light_direction` function:

```
light_direction(light::DirectionalLight, point::Vec3)
light_direction(light::PointLight, point::Vec3)
```

4b. Diffuse Shading

Since our scenes can have multiple lights, we need to calculate the color contribution from each light source. In TODO 4c, we’ll handle lights one at a time by calling `shade_light` in the `determine_color` method that handles `PhysicalShadingModel` shaders. To support this, implement the first method of the `shade_light` function, which calculates the (`Lambertian`) shading constant for a given point and light source, then multiplies that by the point’s diffuse color. Recall that the diffuse color is calculated as:

$$L = k_d I \max(0, n \cdot \ell)$$

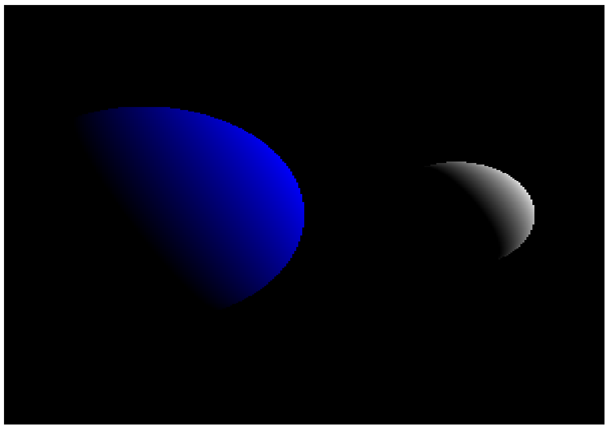
where k_d is the diffuse albedo, I is the light intensity, n is the surface normal, and ℓ is the light direction. You can use the `get_diffuse` function from `Materials.jl` to get the diffuse color of an object; its second argument is a texture coordinate which we aren’t handling yet, but it will behave just fine if you pass `nothing`, so I suggest passing in `hitrec.uv`, which will be `nothing` for now but will eventually contain texture coordinates. You already wrote the `light_direction` function to find $\ell\ell$, and the `Light` object can tell you its intensity.

4c. Determining a pixel’s color under a physical shading model

Now we can implement `determine_color`, calling `shade_light` for each light source. At this point, your raytracer should be able to render objects with Lambertian shading models. Try it out with

```
julia> RayA2.main(2, 1, 300, 300, "results/out_2.png")
```

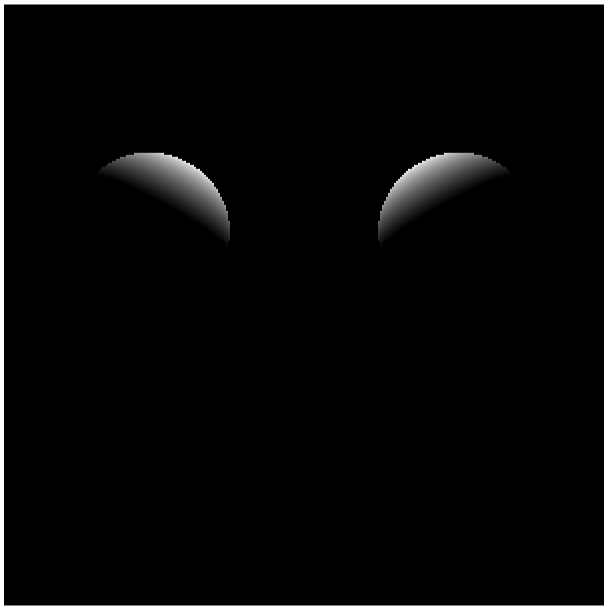
Test Scene 2 uses a directional light source, so if your code for the directional method of the `light_direction` function, `shade_light`, and `determine_color` are correct, you should get the following image:



Test Scene 3 uses a point light light source, so if the `PointLight` variant of the `light_direction` method is implemented correctly, you should be able to run:

```
julia> RayA2.main(3, 1, 300, 300, "results/out_3.png")
```

and get a sinister-looking result that looks like this:



4d. Diffuse-plus-specular shading: Blinn Phong shading model

Next, let’s enable shinier spheres by implementing the Blinn-Phong shading model, which uses a diffuse component just like the Lambertian model, but adds a specular component. Recall that the Blinn-Phong shading equation is :

$$L = k_d I \max(0, n \cdot \ell) + k_s I \max(0, n \cdot h)^p$$

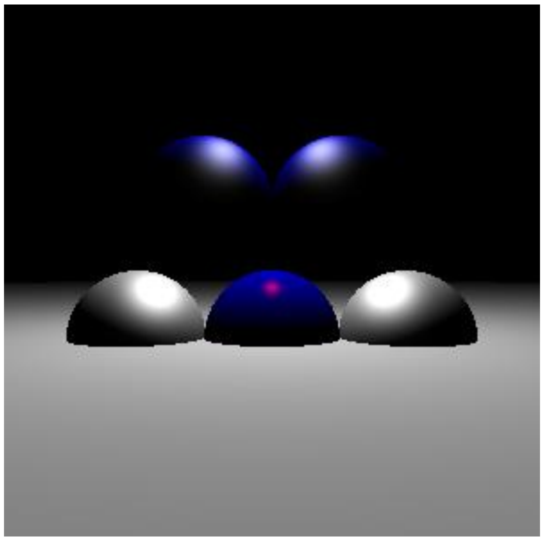
where k_d and k_s are the diffuse and specular colors, n is the surface normal, ℓ is the light direction, I is the light intensity, p is the specular exponent, and h is the half-vector between the viewing and lighting directions, defined as

$$h = \frac{v + \ell\ .}{||v + \ell||}$$

where v is the view direction. The specular color and exponent are stored as part of the `BlinnPhong` shading model struct; the view direction can be determined from information in the `HitRecord`.
If the `BlinnPhong` method of the `shade_light` function is working correctly, you should be able to render Test Scene 4

```
julia> RayA2.main(4, 1, 300, 300, "results/out_4.png")
```

and get the following result:

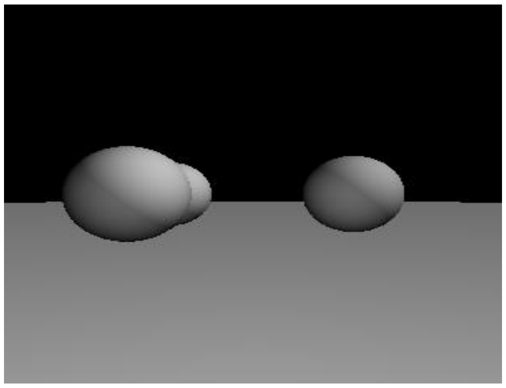


5. Shadows

At this point, if you render Test Scene 5:

```
julia> RayA2.main(5, 1, 300, 300, "results/out_5.png")
```

you’ll get an image that looks like this:



There’s “realistic” lighting here, but one important thing is missing: **shadows**. One of the most awesome things about raytracing is that many physical phenomena of light transport, from shadows to reflections to refraction, are actually pretty simple to do. When we hit an intersection, before we go ahead and compute the contribution from a given light source, all we have to do is first ask: is the light source actually *visible* from this point? Or in other terms, if I shoot a ray from this point towards the light source, does it hit any objects before it arrives at the light?

5a. Check if a point is shadowed from a light

Implement both methods of the `is_shadowed` function. Because `DirectionalLight`s have constant direction, you can think of them as being infinitely far away, so the method for `DirectionalLight`s is the simplest. If a ray from the point towards the light direction hits an object, then the object is in the way of the light source and the point won’t receive light from that source. The only thing to be careful of is the `tmin` and `tmax` values: in a world of

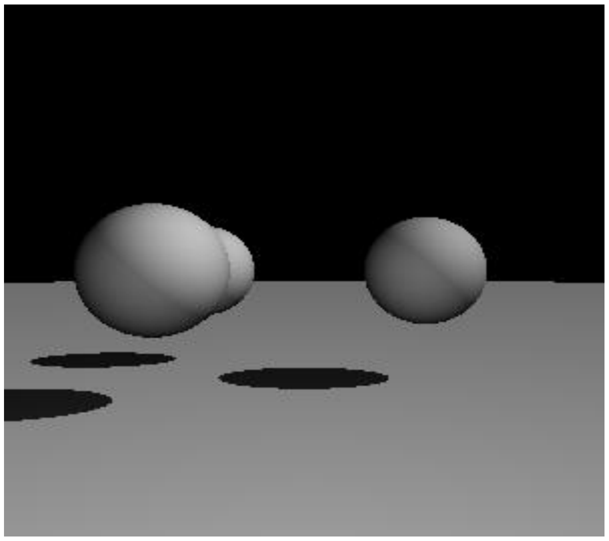
perfect precision, we'd use `0` and `Inf`. However, you have to be careful to avoid hitting the object that you're starting from - so instead of 0, set `tmin` to some small constant, such as `1e-8`.
The `PointLight` method of `is_shadowed` is no more complicated, but the choice of `tmin` and `tmax` is different. Because the light source exists at a point in space, we only want a point on a surface to be shadowed if there's an object between the point and the light source. This requires us to know how far away the light source is. The easiest way to account for this is to set up your shadow ray to have its origin at the surface and its direction vector point directly to the light source (i.e., its length is the distance from the point to the light). This way, the values of `t` we're interested in finding intersections at are simply from the surface to `t=1`, past which we'd be hitting an object beyond the light source which therefore wouldnt' be casting a shadow.

5b. Modify `determine_color`

Now that we've implemented `is_shadowed`, all we need to do is modify `determine_color` to call it. For each light, we now want to check if the point is shadowed with respect to that light, only adding in the contribution from the light source if it's not obscured by another object.

At this point, if we try again on Test Scene 5, we'll see shadows cast on the ground by each of the floating spheres:

```
julia> RayA2.main(5, 1, 300, 300, "results/out_5.png")
```



6. Mirror Reflections

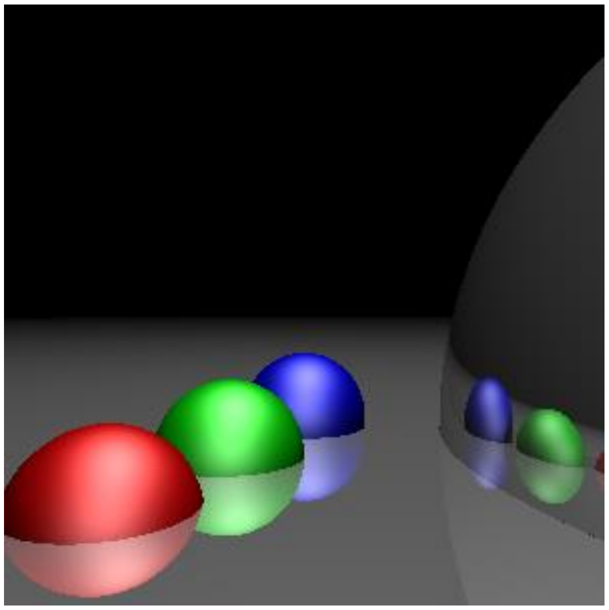
As with shadows, adding mirror-like surfaces is not that complicated. If the surface were 100% reflective (i.e., a perfect mirror), we'd determine at what direction the ray would bounce off, then shoot a new ray in that direction and determine the color of that surface. If that surface happens to have mirror-like properties, well… this sounds like a use case for recursion!

Our renderer will support surfaces that do some of each - they have a surface color shaded according to some already-supported shading model, but that color is mixed with the color of whatever is reflected in the mirror-like surface. Notice that the `Material` struct in `Materials.jl` has a thus-far-ignored field called `mirror_coeff`. This is a value between 0 and 1 that determines what fraction of the color is determined by the local color (this surface's shading model and diffuse color) vs. mirror reflection.
Modify the `traceray` function as follows:

- As before, it computes the color at the surface - this is the *local color*
- If the material of the object has a nonzero `mirror_coeff` value, determine the direction the light will reflect, then recursively call `traceray` to find the `reflected_color` seen in that direction.
- Return the final color, which is `mirror_coeff` times the reflected color and `1-mirror_coeff` times the local color.
- You'll need to respect the recursion depth cap passed into `traceray` to avoid infintely-reflecting mirrors (and also to keep runtime under control). In my implementation I call `traceray` with a `rec_depth` of 8.

At this point, Test Scene 6 should look like this:

```
julia> RayA2.main(6, 1, 300, 300, "results/out_6.png")
```

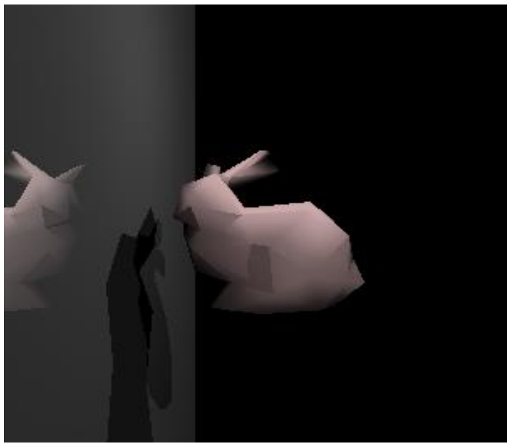


7. Ray-triangle intersection and mesh support

Spheres are great and all, but it would be nice to be able to render more general surfaces, too. In this step, we'll add support for triangle meshes like the ones you generated in A1. The good news is that from the perspective of the renderer, a triangle mesh is simply a big pile of triangles, each of which can be treated as a separate object that needs to be rendered. I've set up the necessary types for you, so all you have to do for this step is implement another method of the `ray_intersect` function to support ray-triangle intersection; the existing rendering pipeline takes care of the rest!
The `OBJMeshes.jl` from A1 is included, so we can load meshes from files and you can drop in your A1 code to generate spheres and cylinders as well. In the second part of `Scenes.jl`, you'll find the types set up for dealing with triangle meshes. Since the OBJ format specifies geometry but not material properties, I've defined a new type called `Triangle` that associates an `OBJTriangle`, the `OBJMesh` it belongs to, and a `material` with which to render it. The `create_triangles` function creates an array of `Triangle`s given an `OBJMesh` and a material.

After this is completed correctly, you should be able to render Test Scene 7 and see our favorite bunny staring contemplatively at itself in a mirror:

```
julia> RayA2.main(7, 1, 300, 300, "out_7.png")
```



8. General Cameras

So far we've been using Test Camera 1, which is a `CanonicalCamera`, defined in `Cameras.jl`. The same file also has a `PerspectiveCamera` type that supports perspective cameras with arbitrary positions, orientations, and focal lengths. It is not fully general in that it still assumes a viewport width of 1 centered on the optical axis (i.e., centered at $(u,v)=(0,0)$) and parallel to the uv plane, so it can't handle shifted or oblique perspective views. To support this camera type, we need two things: a constructor and a `pixel_to_ray` method to generate rays from a `PerspectiveCamera`.

8a. Constructing a `PerspectiveCamera`

To build a `PerspectiveCamera`, we could simply specify the origin and $\vec{u}, \vec{v}, \vec{w}$ axes, but this is cumbersome; usually we want something a little more intuitive. The constructor for `PerspectiveCamera` in `Cameras.jl` takes:

- `eye` - a 3-vector specifying the eye position (center of projection)
- `view` - a 3-vector giving the direction in which the camera is looking
- `up` - a 3-vector giving the direction that is considered "up" from the viewer's perspective. This is not necessarily orthogonal to `view`. Often in scenes defined the way we'd normally think of them, this would be $(0,1,0)$.
- `focal` - a floating-point scalar that specifies how far the image plane is frome the eye

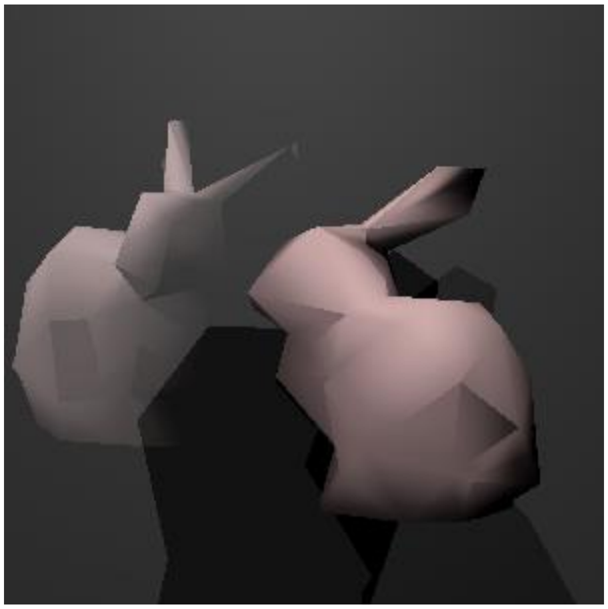
as well as image dimensions. To get $\vec{u}, \vec{v}, \vec{w}$ we need to "square up" a basis from these vectors. See Marschner 4.3 and 2.4.7 for details on this.

8b. Generating rays

Implement the `pixel_to_ray` method for `PerspectiveCamera`s. This step is analogous to the `pixel_to_ray` method for `CanonicalCamera`s; you'll still need to convert pixel coordinates into (u,v) coordinates, but the origin of the rays is now the camera's `eye` point, and the (u,v) are coordinates in terms of the $\vec{u}, \vec{v}, \vec{w}$ basis specifying its orientation.

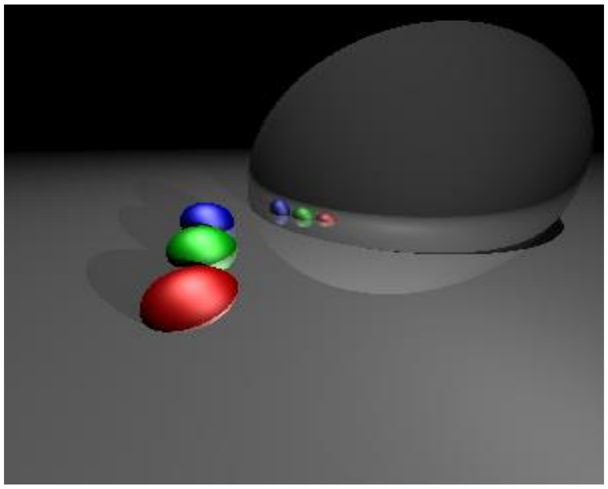
Test Scene 8 is actually the same as Test Scene 7. If we use Test Camera 2, we'll see it from a different perspective:

```
julia> RayA2.main(8, 2, 300, 300, "out_8.png")
```



Camera 3 has a very wide field-of-view (i.e., short focal length), which creates some weird-looking perspective distortion when rendering Scene 6:

```
julia> RayA2.main(6, 3, 300, 300, "out_8b.png")
```



9. Supporting Textures

So far we’ve ignored `texture` field of the `Material` struct. `Materials.jl` includes a type `Texture` that represents some image data to be applied as a texture. This requires changing `get_diffuse` to look up a texture value instead of the object’s diffuse color, and modifying the `ray_intersect` methods to store accurate `uv` coordinates in the `HitRecord`s that they return. Here’s a suggested approach:

9a. Texture Lookups

`Materials.jl` contains a function `get_texture_value` that takes a `Texture` and a `Vec2` of (u,v) coordinates and returns an RGB value at those coordinates. This requires converting the (u,v) coordinates, which are in the range [0,1] into pixel coordinates, which are in the range determined by the image size. This mapping will probably not result in an integer pixel value; there are several ways to deal with this. The simplest is to round to the nearest integer pixel coordinates (nearest-neighbor); the next simplest is to do bilinear interpolation; you can get even fancier with schemes like bicubic and so on.[1] My implementation uses nearest-neighbor.

9b. Modify `get_diffuse`

This one's pretty simple - modify `get_diffuse` to checkwhether a texture is present and call `get_texture_value` if so; otherwise, return the object's diffuse color as before.

9c. Texture coordinates in `ray_intersect` (sphere)

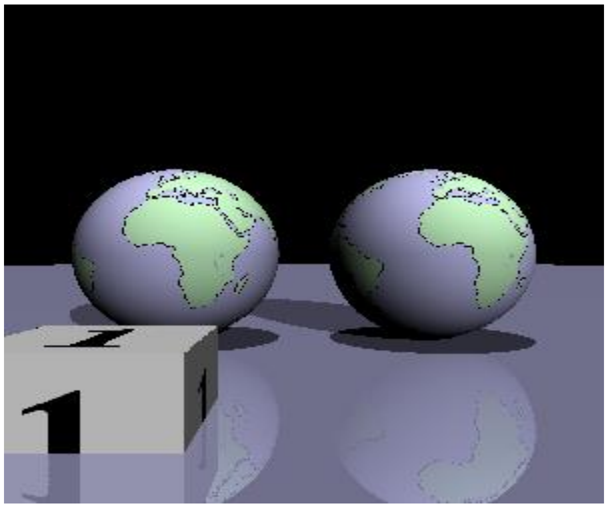
The final step is to make sure that correct texture coordinates are supplied to the above, by filling in the correct value for `uv` in the `HitRecord` objects produced when intersecting rays. Modify the `ray_intersect` method for `Sphere`s to add correct texture coordinates. Use the exact texture mapping function we used in A1. You may find one of the HW1 problems helpful for finding the Sphere texture coordinates.

9d. Texture coordinates

Modify the `ray_intersect` method for `Triangle`s to add correct texture coordinates to the `HitRecord`. Use barycentric interpolation among the texture coordinates at the triangle's three vertices.

If texture coordinates are working for both triangles and spheres, Test Scene 9 should look like this:

```
julia> RayA2.main(9, 1, 300, 300, "results/out_9.png")
```



10. Model your own scene

If you haven't already, take a look at `TestScenes.jl` to see how the scenes you've been rendering so far are set up. In this task, you'll create and render an interesting scene of your own design.
Add the code for your scene to `TestScenes.jl` in a new function called `artifact_<username>(img_height, img_width)`, where `username`. This function should return a 2-tuple of (Scene, Camera) that can be passed into the `main` function to render your scene. Render a 1000-pixel-wide image of your scene and save in `results` as `artifact_<username>.png` (again, replace `<username>` with your username) and commit it to your repository.

If you've implemented the animation extension, your artifact may be a video instead of a still image. In this case, save the video as `artifact_<username>.mp4`.

How and What to Submit

1. Generate a **480x480** render of each of the test images provided in this writeup and commit it to your repository in the `results` directory.
2. **Each group member** should make sure their own scene code is in `TestScenes.jl` and `artifact_<username>.png` is in `results/`.
3. Submit your code by pushing your final changes to Github before the deadline.
4. **Each group member** should fill out the **A2 Hours** poll on Canvas with an estimate of the hours they spent on this assignment.

Extensions

This section may be expanded if more extension ideas arise - stay tuned, and feel free to propose your own.

There are so many cool places to go from here! 580 students should complete at least 2 category-points worth of extra credit. Some of the category 2 ones (combinations or further elaborations on them) point in the direction of possible final projects. If you're unsure about what you're getting into with any of these, please come talk to me. You can also devise your own extensions that are not on this list, but please run them by me first to be sure you're not wasting time.

Before starting on your extensions, please create a separate `extensions` branch in your repository and complete your extensions there. Include your `readme.txt` in both branches. I will use my own modified `TestScenes.jl` for grading purposes, so your `master` branch should contain a working version of the base assignment without any architectural changes.

Category 1

- Implement an `OrthographicCamera` to generate parallel rays from an Orthographic camera and a `GeneralPerspectiveCamera` that supports a viewport that is not centered at (u,v) = (0, 0). In both cases, the camera should also support oblique views by allowing the user to specify the image plane normal separately from the view direction. Play around with these cameras and create images that show the effects of using different camera models. See Tilt-Shift photography for examples of real-life uses for such cameras.
- Our current system is pretty limited when it comes to positioning and manipulating objects. Both types of Scene objects (Spheres and Triangles) currently can only be placed in a particular position; spheres can be scaled (by setting their radius), and the `TestScenes.jl` has a helper method that implements a hacky approach to scaling and translating the position coordinates. Modify Scenes.jl so that the position, scale, and orientation of objects can be specified generally by a transformation matrix. Include helper functions to generate standard transformations to make it easy to model more complex and varied scenes.
- Add support for animnation. This really just means rendering many images in a row with the scene changing a bit between each one. Create tools that make this as easy as possible for the modeler. As an example, you might let a modeler (e.g.,, a person writing code in `TestScenes.jl`) specify a parametric equation that calculates the position of an object, light, or camera as a function of a `time` variable, then generate and render a sequence of scenes over increasing values of `time`. This would be a good one to combine with implementing transformations, since that allows for more flexibility in how objects can move. Create one or more videos showcasing your system's animation capabilities.

- Implement a `SpotLight`, which is like a point light source but only emits light in a cone of directions. The modeler specifies the light's primary direction and the angle of the cone within which light is emitted. The spotlight should support "soft" edges: make the intensity of the light fall off gradually as a function of angle to the spotlight's direction, and include a softness parameter that lets the modeler control how suddenly it falls off.

Category 2

- Implement one of the techniques from Chapter 13:
  - Instancing
  - Distribution Ray Tracing (antialiasing, soft shadows, glossy reflections, area lights, depth of field)
  - Transparency and Refraction
  - Constructive Solid Geometry[2]
- Generalize the texture mapping capabilities of the ray tracer. Specifically, refactor the code to allow `Texture`s to specify more properties than just color. Examples include shading parameters (such as the specular coefficient in Blinn-Phong), normal/bump maps, and displacement maps. See Chapter 11 for more on this.
- From the perspective of physical accuracy, the two shading models we've implemented (Lambertian and Blinn-Phong) are rubbish. They look somewhat plausible, but don't come close to matching up with what surfaces look like in reality. One family of physically-based shading model, known as Microfacet models, is based on a theory of approximating surfaces by random distributions of microscopic flat surfaces (facets), then analyzing the physics of how light interacts with that geometry. You can read up on microfacet models here, and here. For this extension, refactor the shading code to apply a generic shading model that multiplies the cosine term and light intensity by a generic function that dispatches on BRDF type; then implement at least one microfacet BRDF (Beckmann and GCX are good choices) in addition to a Lambertian and Blinn-Phong BRDF and produce a comparison of the results. You'll definitely have to go looking for details on this - come talk to me if you want to do this and need pointers to the right resources.

What to submit

If one group member wants to do extra credit but the other does not, that's fine - just include a note on this in your `readme` and the group member who completed the extensions will get the credit.

If you implement any of these, you must include:

- code that extends the base assignment framework, including test scenes and/or cameras that show off your results included in `TestScenes.jl`
- example results in `results`
- a `readme.txt` file in the base `RayA2` directory with:
  - a description of what you did and where the code can be found
  - instructions for how to reproduce your results
  - a description of the contribution of each group member to the extensions

---

1. You may have seen these terms in video game quality settings - now you know what they mean. ↩
2. For extra awesomeness, implement transparency and refraction *and* CSG, and build physically-plausible glass lenses by intersecting spheres! ↩