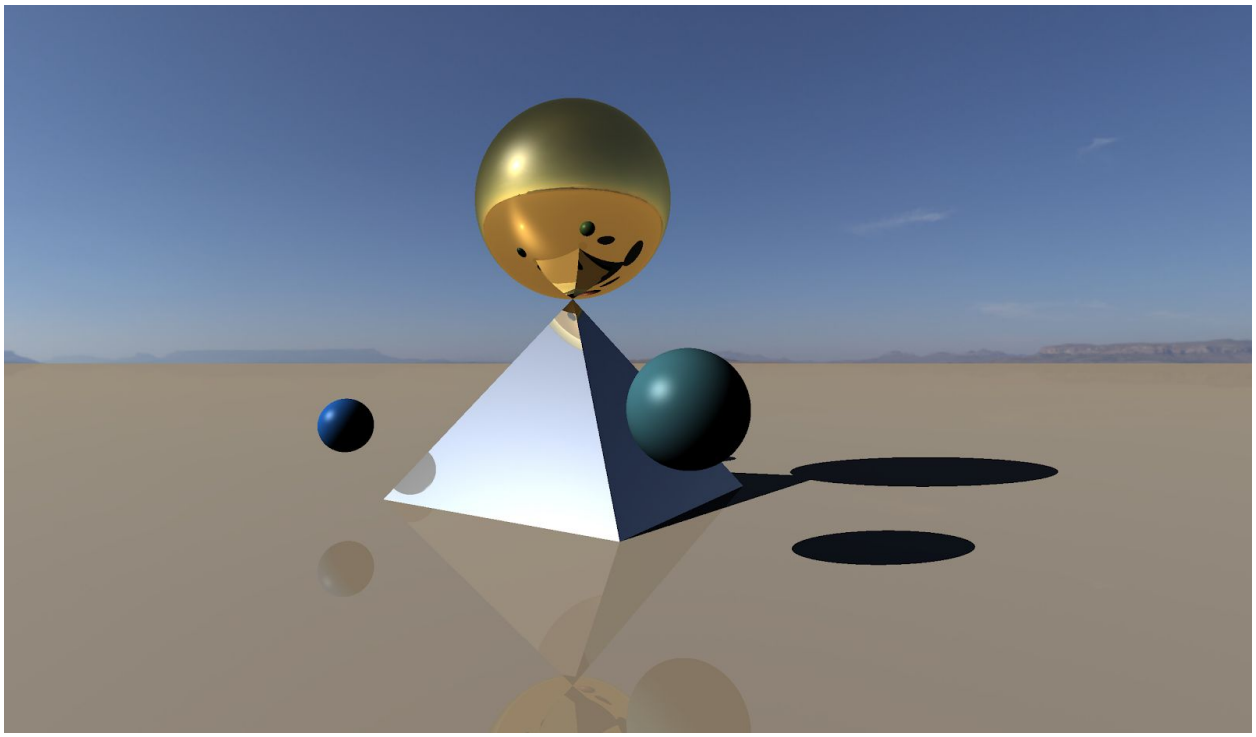
Exercise 5 - Ray Tracing

In this exercise you will write a real-time ray tracer using compute shaders.

The goal of this exercise is to learn about the ray tracing algorithm.

You **must** submit this exercise in pairs.



EX5 Guidelines

- In this exercise you are given the general skeleton of a ray tracer.
- You must complete the missing parts and add features as described.
- Throughout this exercise, colors should be saved in `float` rather than `fixed` vectors to allow for better precision.
- You may add helper functions as needed.

General Guidelines

You may lose points for not following these guidelines.

- Make sure you are using **Unity 2020.1.6f1**
- Make sure that you understand the effect of each part of your code
- Make sure that your code does what it's supposed to do and that your results look the way they should
- Keep your code readable and clean! Avoid code duplication, comment non-trivial code and preserve coding conventions
- Keep your code efficient

Submission

Submit a single `.zip` file containing **only** the following files:

- **RayTracer.compute**
- **Primitives.cginc**
- **Shading.cginc**
- **`readme.txt`** that includes both partners' IDs and usernames. List the URLs of web pages that you used to complete this exercise, as well as the usernames of all students with whom you discussed this exercise

Deadline

Submit your solution via the course's moodle no later than **Sunday, January 24 at 23:55**.

Late submission will result in 2^{N+1} points deduction where N is the number of days between the deadline and your submission. The minimum grade is 0, friday and saturday are excluded.

Part 0 / Setup & Overview

1. Download the exercise zip file from the course Moodle website and unzip it somewhere on your computer.

2. In Unity Hub, go to *Projects* and click the *Add* button on the top right. Select the folder that you have downloaded.

3. Open the project. Once Unity is open, double click the MainScene to open it.

4. The scene contains 3 GameObjects:

    - **Main Camera** - Instead of rendering the scene through the normal Unity rendering pipeline, this camera draws a render texture to the screen after entering play mode.

    - The *Ray Tracer* script component activates the ray tracing compute shader and passes on parameters, such as the camera transform & projection matrix, from the Unity environment (CPU) to the compute shader (GPU).

    - The *Rotate Around* script component rotates the camera around a given point, and is there to help you inspect your raytraced scenes from different angles. Note that you can also edit the parameters of the camera transform directly.

    - **Directional Light** - The light direction is passed on to the ray tracing compute shader and used to light the ray-traced scenes. You can rotate this object and see the light changing.

    - **Sphere** - A sphere GameObject of radius 1 centered at (0, 0, 0). This sphere actually does nothing and will not be rendered to the screen in play mode! It's just there to help you understand the orientation of the camera in relation to the world.

5. Open and inspect the file Ray.cginc.

    - This file defines structs to represent rays, ray hits and materials.

    - There are no classes in HLSL, so we use a pattern of functions that act as initializers for structs, such as `CreateRay` which initializes a `Ray` struct.

6. Open and inspect the file Primitives.cginc.

    - This file defines functions to detect collisions with geometric primitives.

    - Each function receives some parameters that describe the geometry of the surface, as well as 3 basic parameters:

i. `Ray ray` - The ray with which to check for collisions.

ii. `inout RayHit bestHit` - The previously found best hit. If a hit point is found at a smaller distance than the `bestHit.distance`, the function edits `bestHit` to be the new hit point.

    The keyword `inout` allows us to change this parameter's value from within the function.

iii. `Material material` - The material associated with this surface.

7. Open and inspect the file RayTracer.compute, the main module of the ray tracer.

    - All the parameters passed from the CPU are defined here at the top of the file. Take a look and make sure you understand them.

    - After the parameters and constants, we import various modules of the ray tracer. Go over them to get a sense of what each one does.

    - Take a look at the function `CSMain`. This function creates a view ray for each image pixel, calculates the color using the function `trace` then sets the associated pixel color in the render texture. The basic implementation is given to you.

Part 1 / Ray Casting

1. Take a look at the Camera GameObject in the Unity inspector. In the Ray Tracer script component, make sure Scene 1 is selected. This scene contains a single sphere.

   The file Scenes.cginc is not for submission, so feel free to edit it and change the scenes however you like. Scene 0 is empty and you can use it for testing.

2. In the file RayTracer.compute, Implement the function:

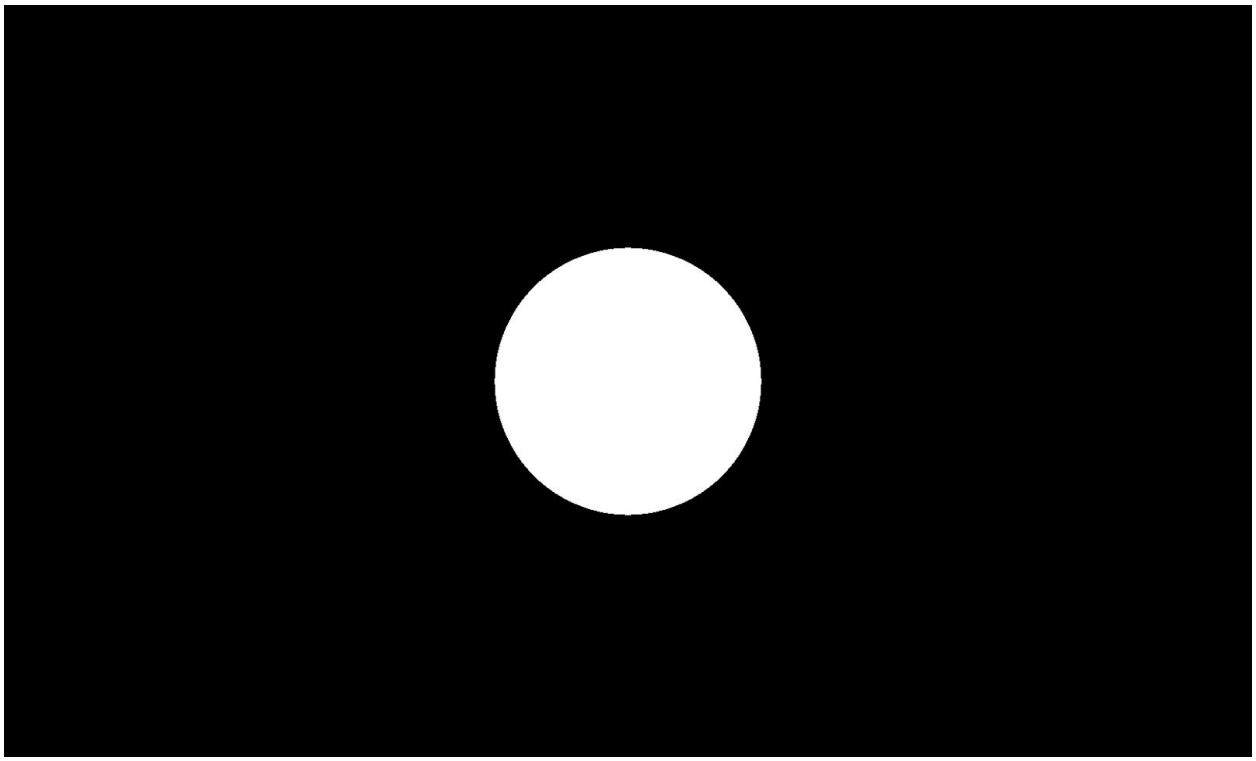
   `float3 trace`

   > The function casts the given `ray` into the currently selected scene using the function `intersectScene`. If the ray has hit something, it shades and returns a color using the function `shadeHit`. Otherwise, it sets the ray's energy to 0 and returns the color given by `shadeMiss`.
   >
   > In any case, remember to multiply the returned color by the ray's original energy.
   >
   > To detect a hit, you must check if `hit.distance` < ∞. You may use the built-in function `isinf` to do so.

3. In the file Primitives.cginc, Implement the function `IntersectSphere` as seen in class. In case of a collision, remember to set all fields of `bestHit` -

   - `float distance` - the distance *t* of the hit point along the ray
   - `float3 position` - the hit point's 3D coordinates
   - `float3 normal` - the surface normal at the hit point
   - `Material material` - the material of the hit surface

4. You should now see an image like this when entering play mode:



Part 2 / Basic Shading

1. Switch the selected scene to Scene 2, which contains a sphere and a floor plane.
2. In the file Shading.cginc, implement the function:

   `float3 blinnPhong`

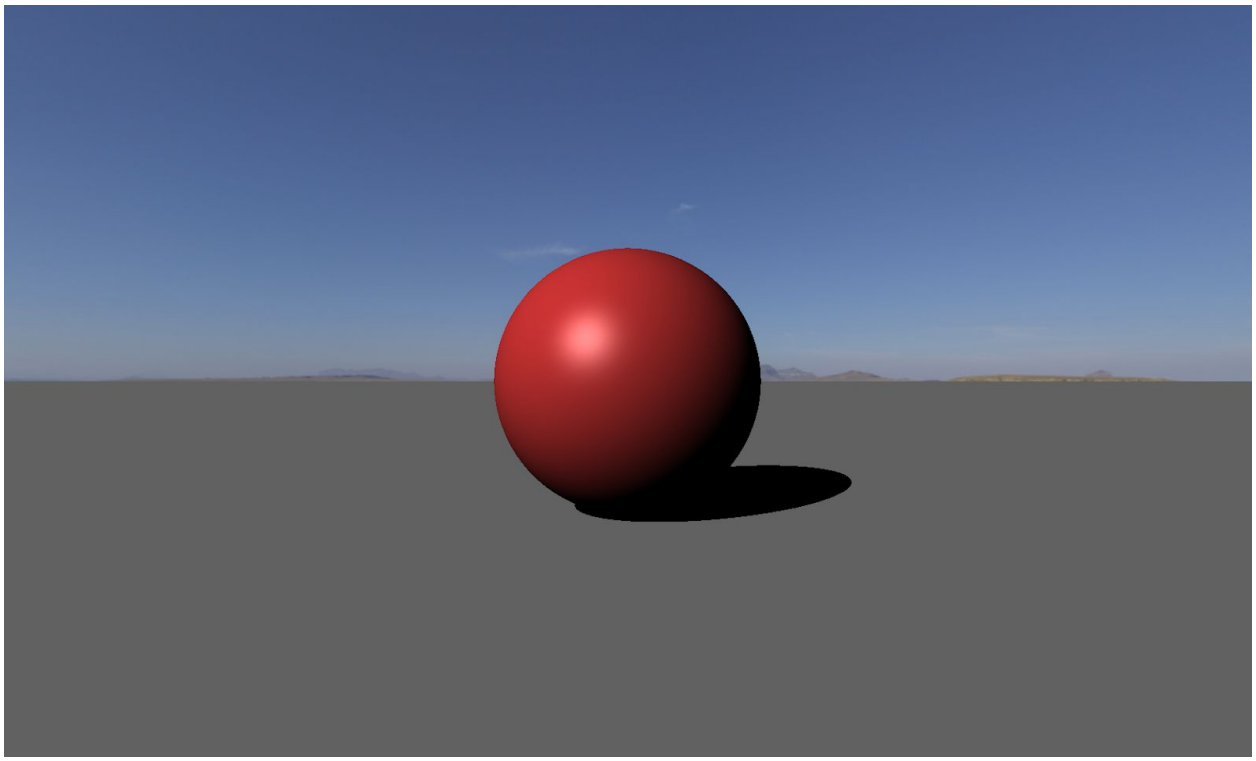
   The function implements an adjusted version of the Phong lighting model, with no ambient component. the diffuse and specular components are defined:

   - Diffuse = max{0, **n** · **l**} * `albedo`
   - Specular = max{0, **n** · **h**}$^{\texttt{shininess} * 0.4}$

   n is the surface normal, **l** is the light direction and **h** is the halfway vector.

   The function returns the sum of the components, Diffuse + Specular.
3. In the file RayTracer.compute, change the function `shadeHit` to use `blinnPhong` when shade each hit point. Hard-code the `shininess` parameter to 50, and set the `albedo` according to the material of the hit point.
4. In the file Primitives.cginc, Implement the function `IntersectPlane` as seen in class.
5. In the function `shadeHit`, cast a shadow ray to implement ray-traced shadows as seen in class.
6. In the function `shadeMiss`, implement environment mapping. You may use the given function `sampleSkybox` to do so.

7. You should now see an image like this when entering play mode:



Part 3 / Reflections

1. Switch the selected scene to Scene 3, which contains various diffuse and reflective objects.

2. In the file Primitives.cginc, Implement the function `IntersectTriangle` as seen in class.

3. In the file Shading.cginc, implement the function:

   ```
   float3 reflectRay
   ```

   The function reflects the given ray from the given hit point. The ray direction and position are calculated as seen in class, while the ray energy is multiplied by the specular coefficient of the hit material k_s.

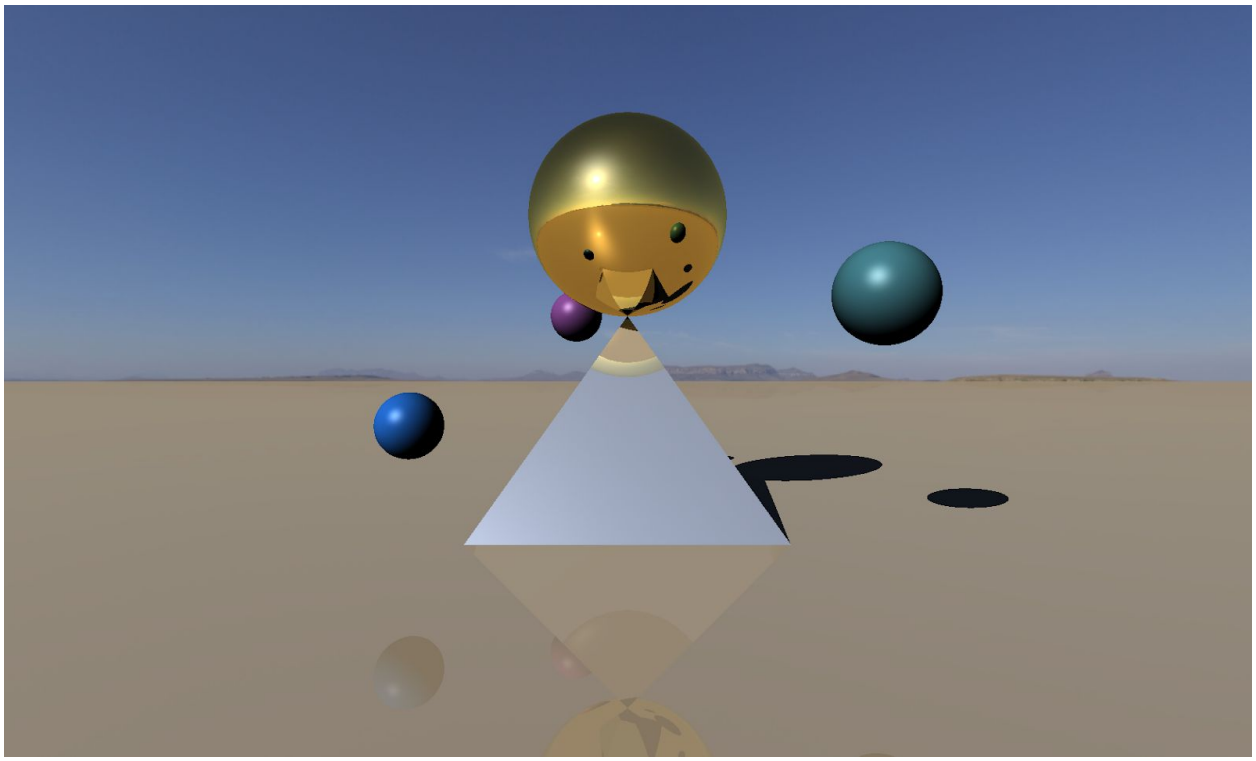
4. In the file RayTracer.compute, in the function `trace`, use `reflectRay` to change the given ray such that it is reflected from the hit point (only in case of a hit, if the ray missed all geometry there is no need to do anything).

5. Change the function `CSMain` so that it will loop `_BounceLimit` times and call the function `trace`, each time with the same ray struct. Remember that inside the function `trace` we are editing the ray, so each iteration will trace a different part of the ray's journey throughout the scene.

   Add to the result the color returned from `trace` at each iteration, so that the final result is the sum of all colors along the ray's journey.

   If the ray has no more energy, we can safely stop tracing it. You may use the function `any` to check this condition, and break the loop using the keyword `break`.

6. You should now see an image like this when entering play mode:



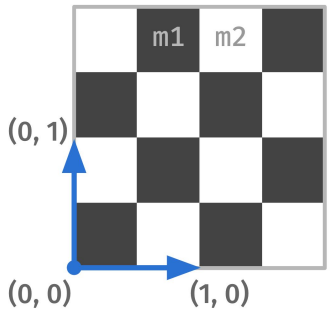
Part 4 / Refractions

7. Switch the selected scene to Scene 4, which contains diffuse, reflective and refractive spheres.

8. In the file Primitives.cginc, Implement the function:

   ```
   void intersectPlaneCheckered
   ```

   Checks for a collision between a ray and a plane. In case of a hit, the material returned is either `m1` or `m2` to create a checkerboard pattern like so:

   The side length of each square in the pattern should be 0.5.

   You may assume the plane is axis-aligned (i.e. oriented along the XY XZ or YZ planes)

   In your readme.txt file, briefly explain how you implemented this function.

9. In the file Shading.cginc, implement the function:

   ```
   float3 refractRay
   ```

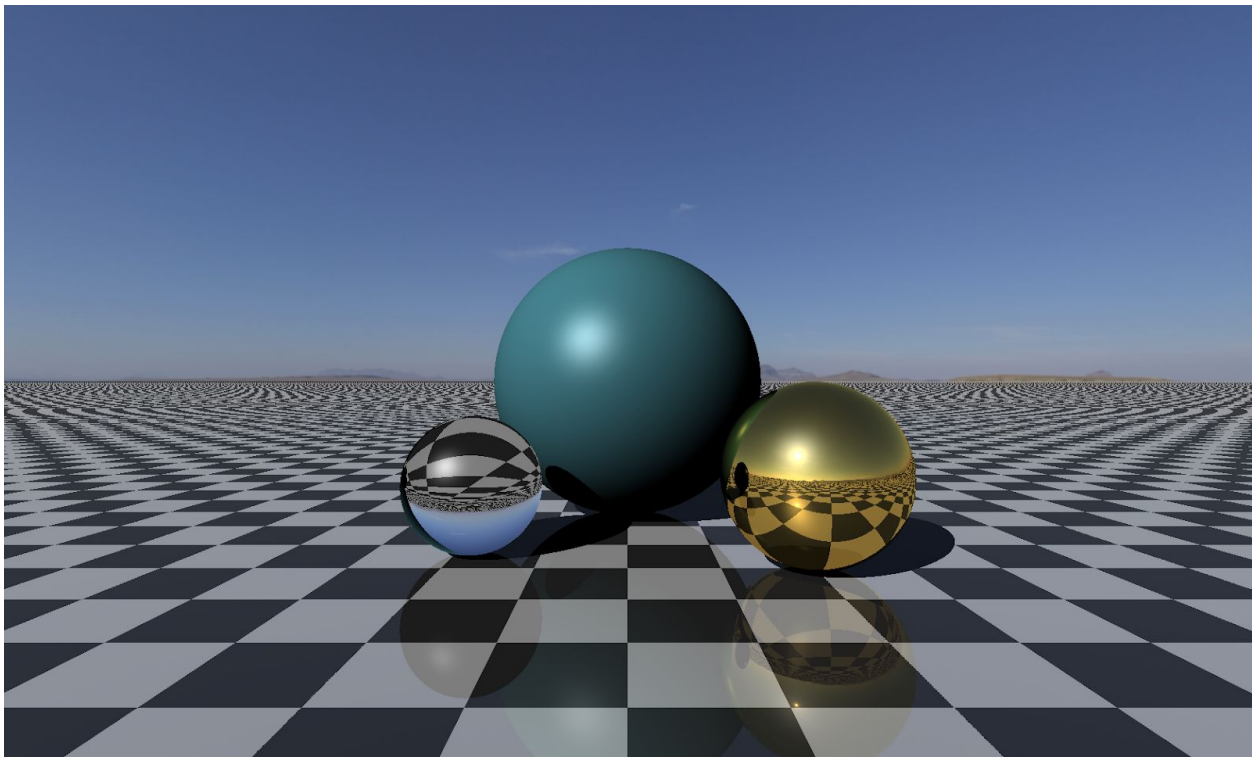
   The function refracts the given ray from the given hit point. The ray direction and position are calculated as seen in class, the ray energy should remain unchanged.

   You may assume that the refractive index of the air (i.e. the space between objects) is exactly 1. You may also assume that the ray is always entering or leaving air, meaning that there can't be a transition from glass to water etc.

10. In the file RayTracer.compute, in the function `trace`, check if the hit material is reflective or refractive - a reflective material will have a refractive index of -1.

    Use `refractRay` or `reflectRay` to bend or bounce the light according to the material. Note that in both cases, `shadeHit` should also be used to shade the hit point.

11. You should now see an image like this when entering play mode:



Part 5 / Cylinders

1. Switch the selected scene to Scene 5, which contains a circle surrounded by cylinders.
2. In the file Primitives.cginc, Implement the function:

   `void intersectCircle`

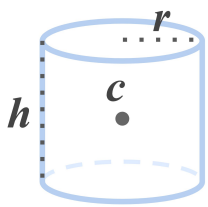
   Checks for a collision between a ray and a circle. The circle is defined by a center center point c, a radius r and an orientation vector n. The orientation vector is the surface normal of the plane on which the circle lies.

   You should now see a golden circle at the center of the scene.
3. In the file Primitives.cginc, Implement the function:

   `void intersectCylinderY`

   Checks for a collision between a ray and a cylinder aligned along the Y axis.

   The cylinder is defined by a center point c, a radius r and height h:

   The implicit representation of a Y-aligned cylinder of infinite height, centered at $c = (c_x, c_y, c_z)$ is given by:

   $$f(p) = f(x, y, z) = (x - c_x)^2 + (z - c_z)^2 - r^2 = 0$$

   You may use the function `intersectCircle` to create the top and bottom caps for the cylinder.

   In your readme.txt file, briefly explain how you implemented this function.

4. You should now see an image like this when entering play mode: