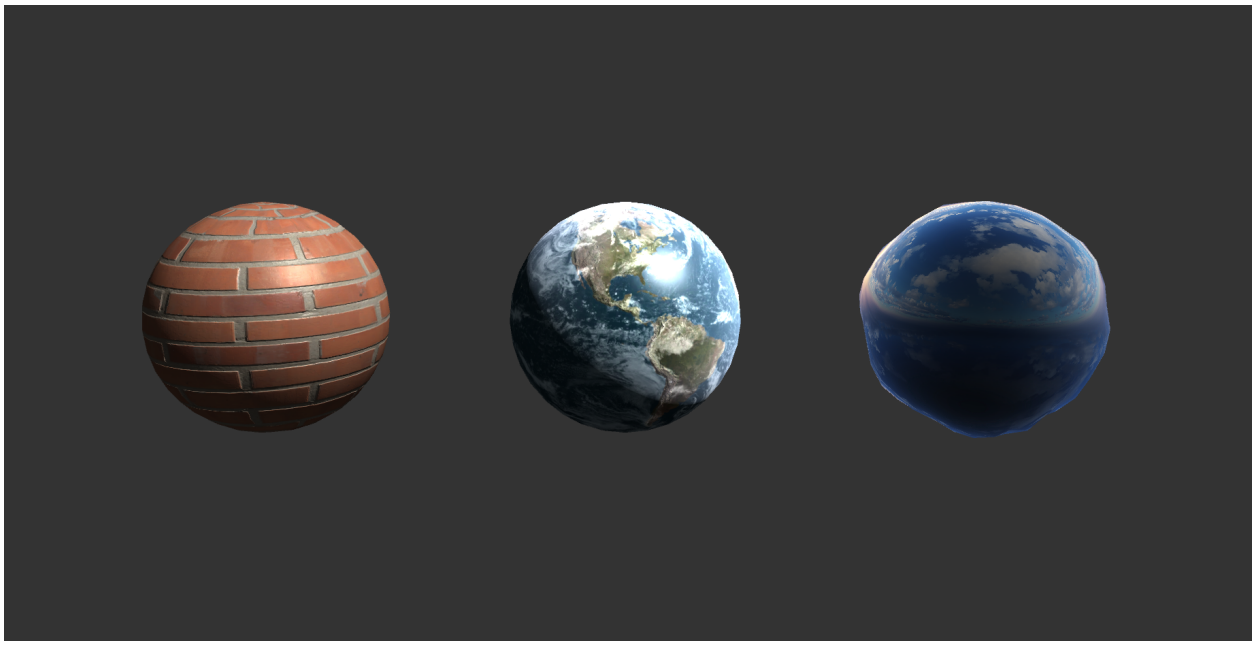
Exercise 4 - Textures

In this exercise you will create different materials using texturing techniques you have seen in class.

The goal of this exercise is to learn about shaders, texturing, UV mapping, displacement mapping and more.

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



EX4 Guidelines

In all shader files:

- You may assume that lighting is directional, i.e. it is infinitely far away so its direction is constant throughout the scene.
- You may not change the ShaderLab properties given in the file.
- You may not change the vertex input structs given in the file, but you can add varyings as needed.
- You may add helper functions as needed.
- You may not use any function from `UnityCG.cginc`, but you may use any built-in variable.

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:

- **Bricks.shader**
- **Earth.shader**
- **Water.shader**
- **CGUtils.cginc**
- **CGRandom.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
- If you choose to do the bonus, include the Bonus folder as described in part 4.

Deadline

Submit your solution via the course's moodle no later than **Sunday, January 15 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, saturday is excluded.

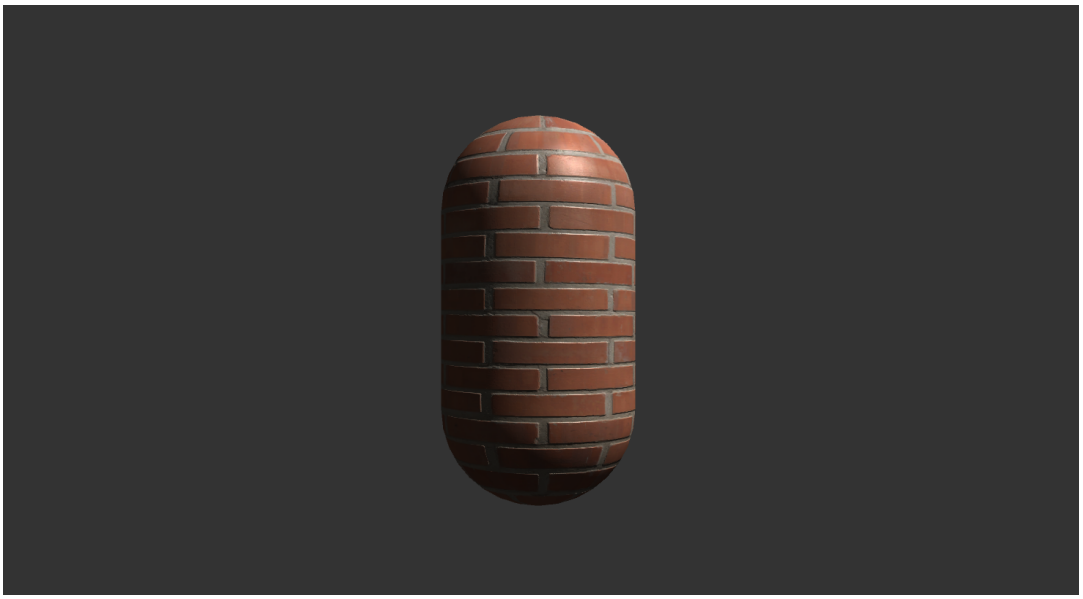
Part 0 / Setup

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.

- Note that there is a scene for each part of the exercise, in the folder Scenes.
- The folder Shaders contains the shader code and helper files containing functions & constants which you may use. You will also need to complete the implementation of some functions in these files.
- The folder Materials contains materials, one per shader.
- The folder Textures contains texture image files.

Part 1 / Bricks

In this part you will write a shader that gives the appearance of a brick wall, using albedo mapping, specular mapping and bump mapping.



1. Open the scene “Bricks” in Unity, then open the file Bricks.shader to edit it.
2. For each fragment, sample the albedo map at the corresponding UV coordinate, given from the `appdata`. The material should now look like this:

3. We will now implement an adjusted version of the Blinn-Phong lighting model to shade the object. In the file CGUtils.cginc implement the function:

```
fixed3 blinnPhong
```

The function implements an adjusted version of the Phong lighting model. The ambient, diffuse and specular components are defined:

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

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

  The function returns the sum of the components, Ambient + Diffuse + Specular.

4. In the file Bricks.shader, use the function `blinnPhong` to shade each fragment.

   - Calculate the n, h, l directions as needed
   - Pass on the _Shininess material property as the shininess parameter
   - Sample the albedo map at (u, v) to get the albedo value
   - Sample the specular map at (u, v) to get the specularity value
   - Pass on the _Ambient material property as the ambientIntensity parameter

   The material should now look like this:

5. Now we will use the heightmap to bump-map the surface and create the illusion of raised bricks. In the file CGUtils.cginc, implement the function:

   ```
   float3 getBumpMappedNormal
   ```

   The function receives a `bumpMapData` struct and returns the world-space bump- mapped normal for the given data.

`struct` **`bumpMapData`**

`float3 normal` - Mesh surface normal at the point (world-space)

`float3 tangent` - Mesh surface tangent at the point (world-space)

`float2 uv` - UV texture coordinates of the point

`sampler2D heightMap` - Heightmap texture to use for bump mapping

`float du` - Increment size for u partial derivative approximation

`float dv` - Increment size for v partial derivative approximation

`float bumpScale` - Bump scaling factor

6. In the file Bricks.shader, use the function `getBumpMappedNormal` to shade each fragment.

   - Use the size of the _HeightMap texture texels as *du* and *dv*. Note that the texel may not be square!
   - Use the _BumpScale material property, divided by 10000 as the bumpScale parameter.

   Finally, exchange the normal used in step 4 to calculate the lighting with the new bump-mapped normals:

Part 2 / The Blue Marble

In this part you will write a shader to make a simple sphere look like earth, using spherical texture mapping.

1. Open the scene “Earth”. First we will use the albedo map to color our sphere. To do that, we will use spherical UV mapping as seen in class. In the file CGUtils.cginc implement the function:

   ```
   float2 getSphericalUV(float3 pos)
   ```

   - The function receives a point `pos` in 3D cartesian coordinates (x, y, z)
   - The function returns coordinates (u, v) $\in [0, 1]^2$ corresponding to `pos` using spherical texture mapping

2. In the file Earth.shader, in the fragment shader function, sample the albedo map at the coordinates you found and color each fragment accordingly. The sphere should now look like this:

A real specular highlight (known as sunglint) reflecting off the ocean, as seen from the International Space Station. Note that the landmass scatters incoming sunlight diffusely.

3. Use the function `blinnPhong` from CGUtils.cginc that you have implemented in part 1 to shade each fragment. The parameters you must send to this function are exactly as described in part 1.

   Tip: You are not given vertex normals in this part. Calculate the normal per fragment with the information that you have.

   The sphere should now look like this:

4. Now we will use the heightmap to bump-map the surface and create the illusion of raised mountain ranges. Use the function `getBumpMappedNormal` from CGUtils.cginc that you have implemented in part 1 to calculate the bump mapped normal for each fragment.

    - Use the mesh surface normal you calculated in step 3.
    - Note that you are not given the vertex tangents in this part. You can calculate the surface tangent by taking the cross product of the normal and the up vector (0, 1, 0).
    - Use the size of the _HeightMap texture texels as *du* and *dv*. Note that the texel may not be square!
    - Use the _BumpScale material property, divided by 10000 as the bumpScale parameter.

    Water bodies on earth are generally flat, so we don't want bump mapping to affect these areas. We have the specular map that tells us which areas correspond to water, so we can use it in the following formula:

    finalNormal = (1 – _SpecularMap[u,v]) * bumpMappedNormal
    + _SpecularMap[u,v] * baseSurfaceNormal

    Finally, exchange the normal used in part 3 to calculate lighting with the final bump-mapped normal. The sphere should now look like this:

    The difference might be a bit hard to see - play around with the bump scale property and take a look at the mountain ranges around the world.

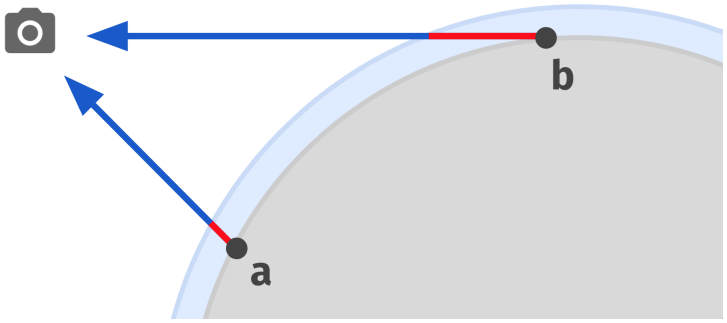
5. Now we will simulate the atmosphere. This part will consist of atmospheric scattering and clouds. First we will define the Lambert (diffuse) lighting at each point:

    Lambert = max{0, n · l}

*We do not want the height of the landmasses to affect the atmosphere color, so we will use the basic, non-bump-mapped normal n.

The earth's atmosphere scatters blue light, and so the sky appears blue during the day. The same effect can be seen from space - so we should add it to our shader.

When seen at an oblique angle, we see a "thicker" slice of the atmosphere and so more blue light is scattered into our eyes. We can simulate this effect using the dot product of the surface normal n and our view direction v.



Light coming from point b passes through a thicker layer of atmosphere than light coming from point a, and so point b will appear bluer.

The atmosphere and clouds component definitions:

- Atmosphere = (1 – max{0, n · v}) * sqrt(Lambert) * _AtmosphereColor
- Clouds = _CloudMap[u,v] * (sqrt(Lambert) + _Ambient)

Add these components to each fragment color. The fragment color should now be set to: BlinnPhong + Atmosphere + Clouds.

Finally, The sphere should now look like this:

Part 3 / Noise

In this part you will implement value noise and perlin noise which you can later use to generate procedural textures.

1. Open the scene “Noise”, then open the file CGRandom.cginc. In this file you are given a few functions to generate pseudo-random values, as seen in class. You are also given a function that performs bicubic interpolation, make sure you understand how it works.

2. Inspect the file Noise.shader. Note that this shader is already implemented and it is given to help you implement the noise functions in CGRandom.cginc.

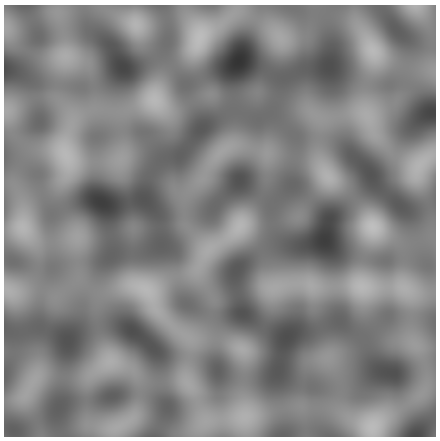
3. In the file CGRandom.cginc, implement the function `value2d`. The function returns a pseudo-random `float` in the range [-1, 1], given by the value noise algorithm at the sampling coordinates `c`.

   Hint: Find the 4 corners of the grid cell containing `c`, then use their coordinates to sample the given function `random2`. Note that it returns a random `float2` - for this part you can just use the first value of the random vector and ignore the second.

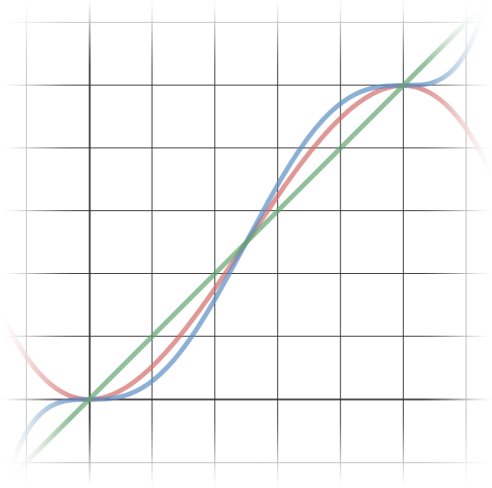
   Finally use bicubic interpolation to calculate the color. You should see a result like this:

4. In the file CGRandom.cginc, implement the function `perlin2d`. The function returns a pseudo-random `float` in the range [-1, 1], given by the Perlin noise algorithm at the sampling coordinates `c`.

   You should see a result like this:



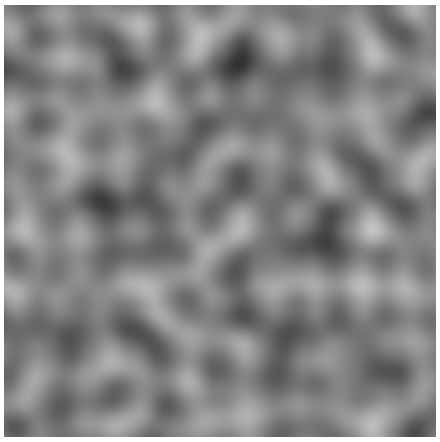
5. We would now like to use better interpolation to achieve a smoother result. Quintic interpolation is similar to cubic interpolation, but uses a fifth-degree polynomial basis rather than a cubic polynomial basis. Cubic interpolations are C^1 continuous - the curve has continuous slope, while quintic interpolations are C^2 continuous - the curve has continuous slope as well as continuous curvature.



- Linear “curve”: $f(x) = x$
- Cubic Hermite curve: $f(x) = 3x^2 - 2x^3$
- Quintic Hermite curve: $f(x) = 6x^5 - 15x^4 + 10x^3$

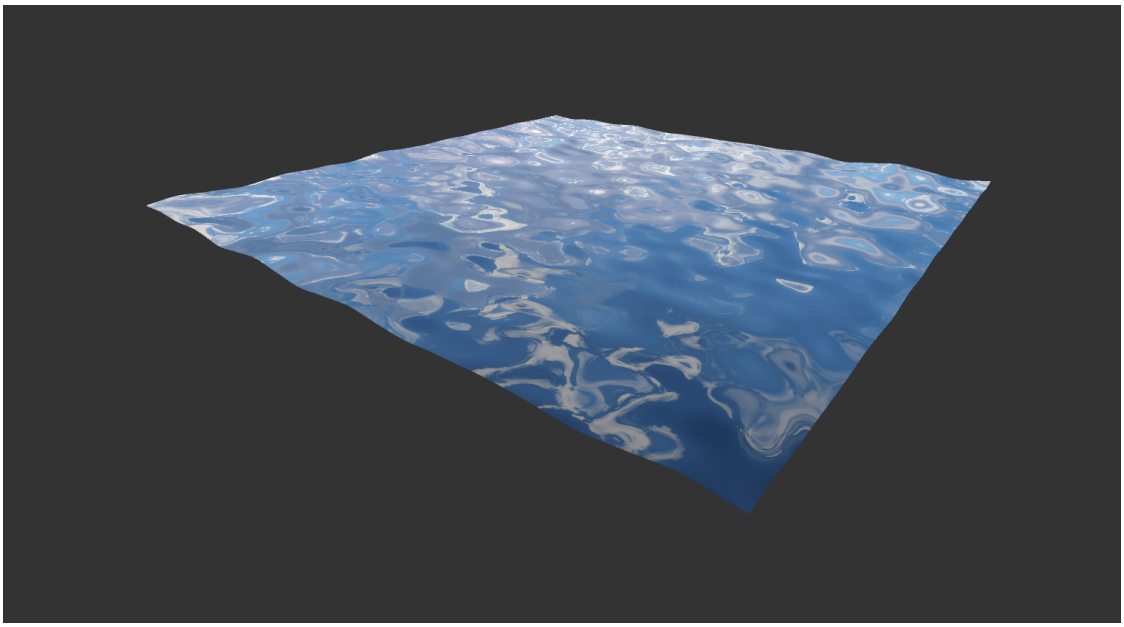
Implement the **`biquinticInterpolation`** function. It interpolates in the x and y directions, similar to the given function `bicubicInterpolation`, but uses the Quintic Hermite curve rather than the Cubic hermite curve.

Finally, use this function to interpolate the influence values in `perlin2d`. You should see a result like this:



Part 4 / Water

In this part you will write a shader to make a simple plane look and behave like water. You will do this using procedurally generated Perlin noise, displacement mapping and reflection mapping.



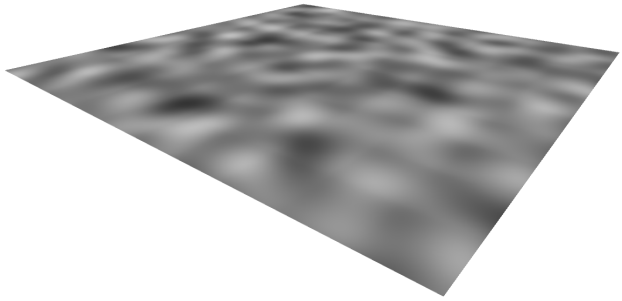
1. Open the scene “Water”, then open the file Water.shader. implement the function:

   ```
   float2 waterNoise(float2 uv, float t)
   ```

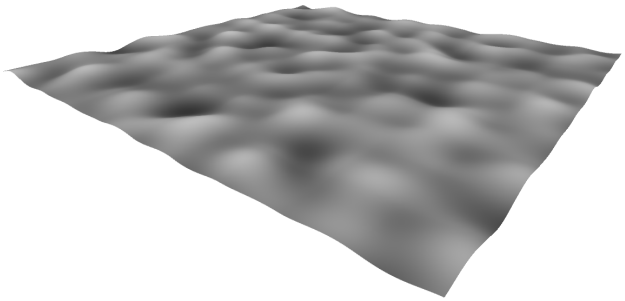
   - The function samples a noise function simulating water, at the given `uv` coordinates and given time `t`.
   - For now, we will ignore the time and just return 2D perlin noise sampled at the given UV coordinates.

2. In the fragment shader, sample `waterNoise` using the UV coordinates multiplied by the _NoiseScale property. You can send 0 as the time parameter. Set the color of each fragment to the noise value, normalized to [0,1].

The material should now look like this:



3. We will now use the generated noise as a displacement map. In the vertex shader, also sample the 2D perlin noise using the UV coordinates multiplied by the _NoiseScale property. Use this value, multiplied by _BumpScale, to displace each vertex position. The material should now look like this:



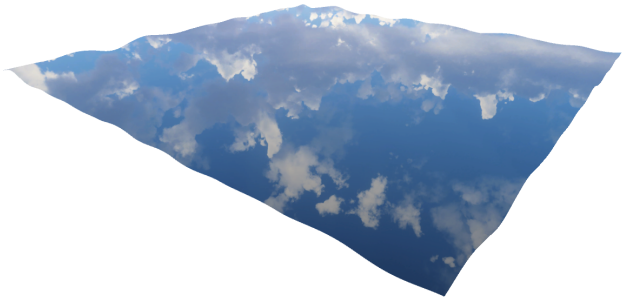
Note that the lower areas of the mesh correspond to the darker areas of the texture.

4. We will now use reflection mapping to simulate the sky reflecting on the water. For each fragment, calculate the reflected view direction r, and use that to sample the given _CubeMap. We'll call the resulting value the *ReflectedColor*.

   At a certain angle (the *critical angle*) light will travel into the water rather than reflecting off of its surface. To simulate this, set each fragment color to the following formula:

   Color = $(1 - \max\{0, n \cdot v\} + 0.2) * \text{ReflectedColor}$

The material should now look like this:



5. As you can see, the reflection looks like a flat mirror and ignores the “waves” we have created. To fix this, we must calculate the wave’s effect on the surface normals. Implement the function:

   ```
   float3 getWaterBumpMappedNormal
   ```

   The function receives a `bumpMapData` struct and time `t`. It returns the world-space bump-mapped normal for the given data and time.

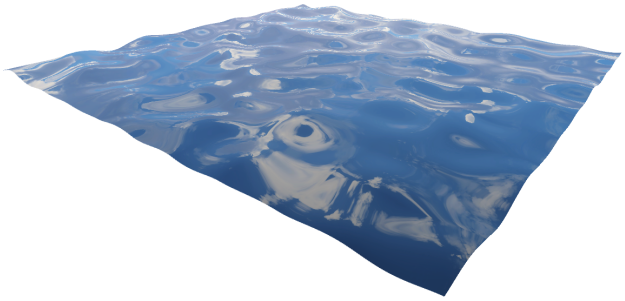
   Instead of using a heightmap texture, we will use a procedurally generated noise function as the heightmap. Ignore the `heightMap` parameter of the `bumpMapData` and sample the function `waterNoise` directly at the given uv coordinates to calculate the normals.

6. Use the function `getWaterBumpMappedNormal` to calculate the new normal at each fragment.

   - Use the constant `DELTA` as both *du* and *dv*.
   - Use the _BumpScale material property as the bumpScale parameter.
   - No need to fill in the `heightMap` parameter of the `bumpMapData`
   - For now, ignore the time parameter `t`.

   Finally, exchange the normal used in step 5 to calculate the reflection with the new bump-mapped normals.

The material should now look like this:



7. We have a material that looks like water, but it is frozen in time! To animate it, we must first implement 3D Perlin noise.

   Open CGRandom.cginc to edit it. Implement the 3D version of Perlin noise in the function **`perlin3d`**. To interpolate between the influence values in 3D, you must also implement the function **`triquinticInterpolation`** as well.

8. Back in Water.shader, change the implementation of `waterNoise` to the following formula, which combines a few layers of Perlin noise at different scales and speeds:

   Perlin3D(0.5u, 0.5v, 0.5t) + 0.5 * Perlin3D(u, v, t) + 0.2 * Perlin3D(2u, 2v, 3t)

9. To get time information in a shader, we can use the built-in variable `float2 _Time` provided by Unity. This variable contains the time since level load `t`, pre-multiplied by different scales: `(t/20, t, t*2, t*3)`. We will just use the regular time, given in `_Time.y`.

   At every call to `waterNoise`, send the time `t` multiplied by _TimeScale. You should now see the water animating smoothly!

Part 5 / Bonus

Hopefully by now you can see some of the possibilities that shaders and textures give us in computer graphics. As a bonus, create a cool material of your choice!

You may use any of the tools that we learned or something else entirely. You can design your material for any of Unity's default meshes, or one of the meshes included in the Models folder.

In your submission zip file, don't forget to include the Bonus folder, including:

- A screenshot or screen recording of your bonus material
- The Bonus scene
- The file Bonus.shader
- The material Bonus.mat
- Any other assets you need (e.g. texture image files)

Note: your bonus must include some *interesting* changes to the shader code. Just changing the textures or colors for one of the existing materials will not be accepted.

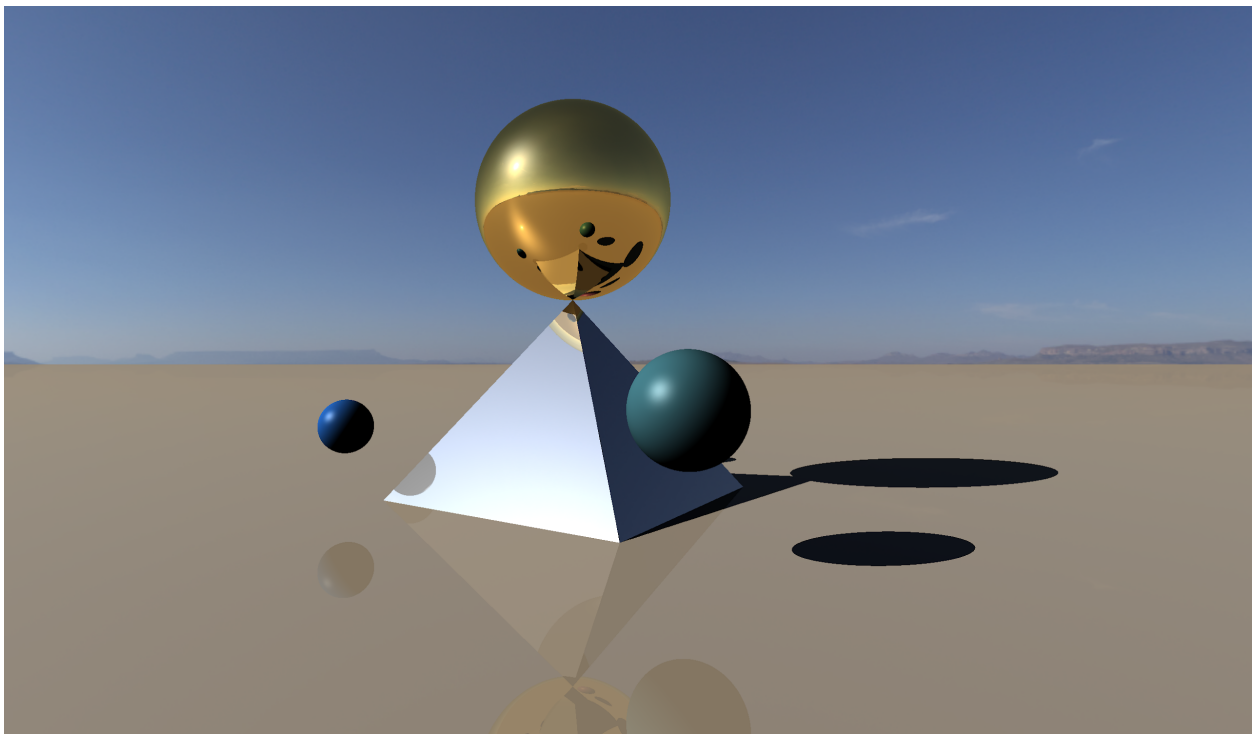
Exercise 5 - Ray Tracing

CIRCLE + TRIANGLE SHADOWS

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 16 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, saturday is 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:

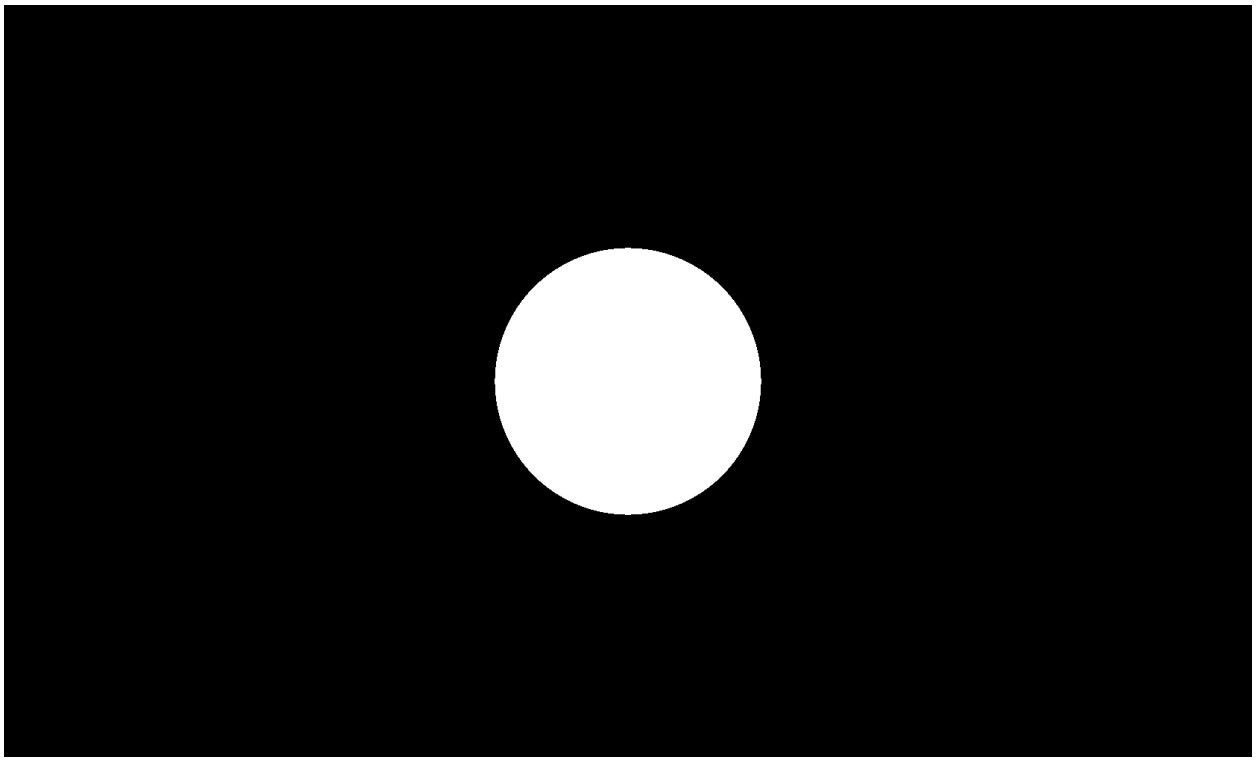
   `float3 trace`

   > 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`

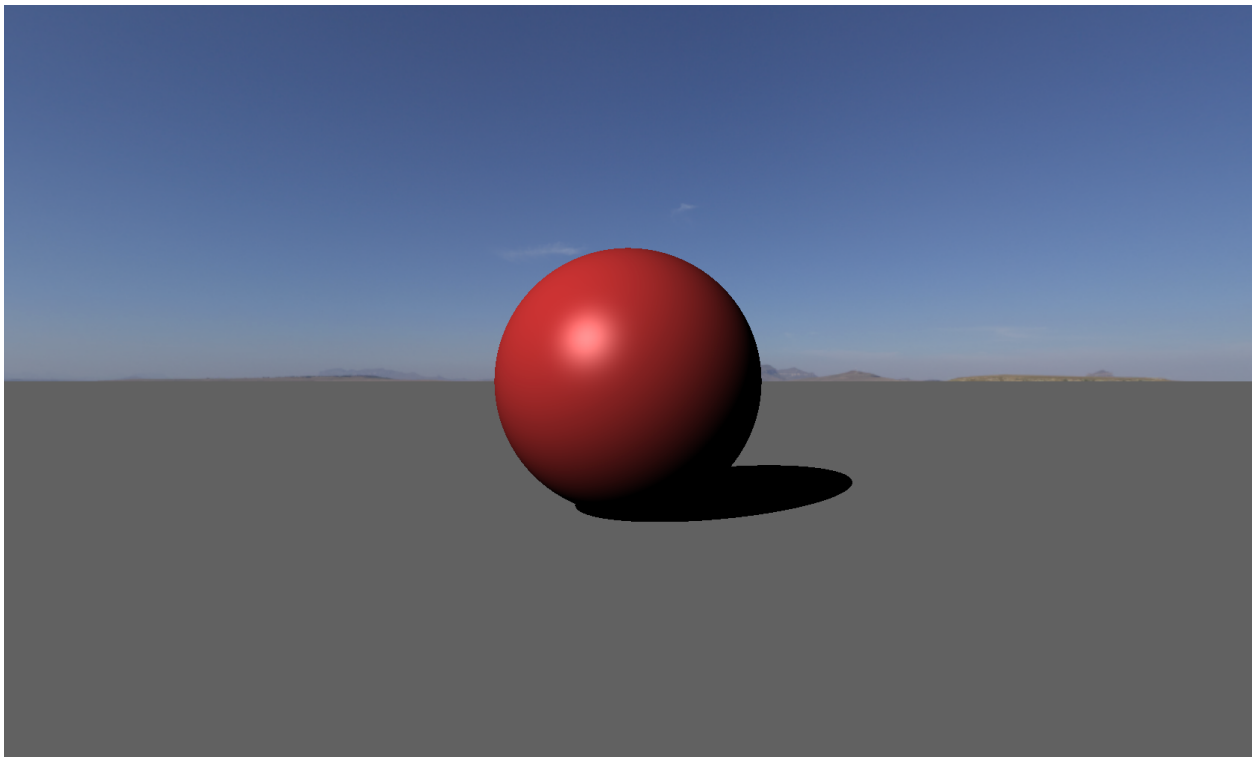
   The function implements an adjusted version of the Phong lighting model, with no ambient component. the diffuse and specular components are defined:

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

   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`.