# Monday

## 6:55-7:10, 7:25-7:35 Fill out today’s work log

**6:55** Added planning to the drop-down menu of the types of tasks.

**7:00** Arrived at switch station.

**7:25** I can’t plan more tasks for the work log. I first need to know my higher level goals.

## 7:35-8:00 Fill out previous sprint’s learning log

**7:35** Finished writing about feedback

**7:40** Reflected on ILOs

**7:45** Reflect on notes taken during week 5

**7:55** Postpone task to after meeting

It feels more important to prepare material to discuss with the teachers right now. Especially after having read the ILOs

## 8:00-8:15 Prepare questions for meeting with David

**8:00** Prepare some questions

* I have a feeling that I’m cheating a bit. I’m reading about all these articles and formulas people have made up. All the information is very easy to find thanks to the internet. Many people write about their mistakes and oversights. I find it quite easy and very useful to learn from their mistakes.   
  Every time that I find something like this, I feel like I’m cheating a bit. It is something that is so obvious, but I wouldn’t have been able to come up with it myself. An example of this is almost every formula that I have used in the program. Take the plane intersection formula for example. I wouldn’t have come up with the fact that when you dot the directional vector with the plane normal and the result becomes 0, the lines don’t intersect and else they do. It’s something that I never thought of before, but now that I know it I can’t think of a different way.  
  Isn’t our job to translate abstract concepts into code. Should I not be able to observe this problem (the intersection of a plane) and come up with a solution myself? Is it alright if I take someone’s else his idea and apply it to my own project?  
  I know that programmers are advised to reuse code. It is tested thus should contain fewer errors and it makes us efficient. So my question is, is it alright if I just understand the solution even though I didn’t come up with it myself?  
  I have answered my own question actually. I think that the teachers once told me that it’s alright if you understand what you are doing. If you are blindly copying things and hoping that it works, it isn’t alright.

## 8:45-10:40 Fill out previous week’s sprint learning log

## 10:40-10:45 Fill out this sprint’s higher level goals

## 10:45-10:55 Fill out today’s work log

## 10:55-11:45 Update json file structure

Stoi parses to an integer. An integer is 4 bytes large. We use unsigned integers. Integers on a 64 bit machine should be 4 bytes but because the value is an integer, it is out of bounds. To assure that I have 4 bytes, I should be using long. Long is assured to have 4 bytes which is exactly what we need and we have a function for it called stoul.

## 12:15-16:30 Refactor project and implement reflection

**12:15** Start from a clean project

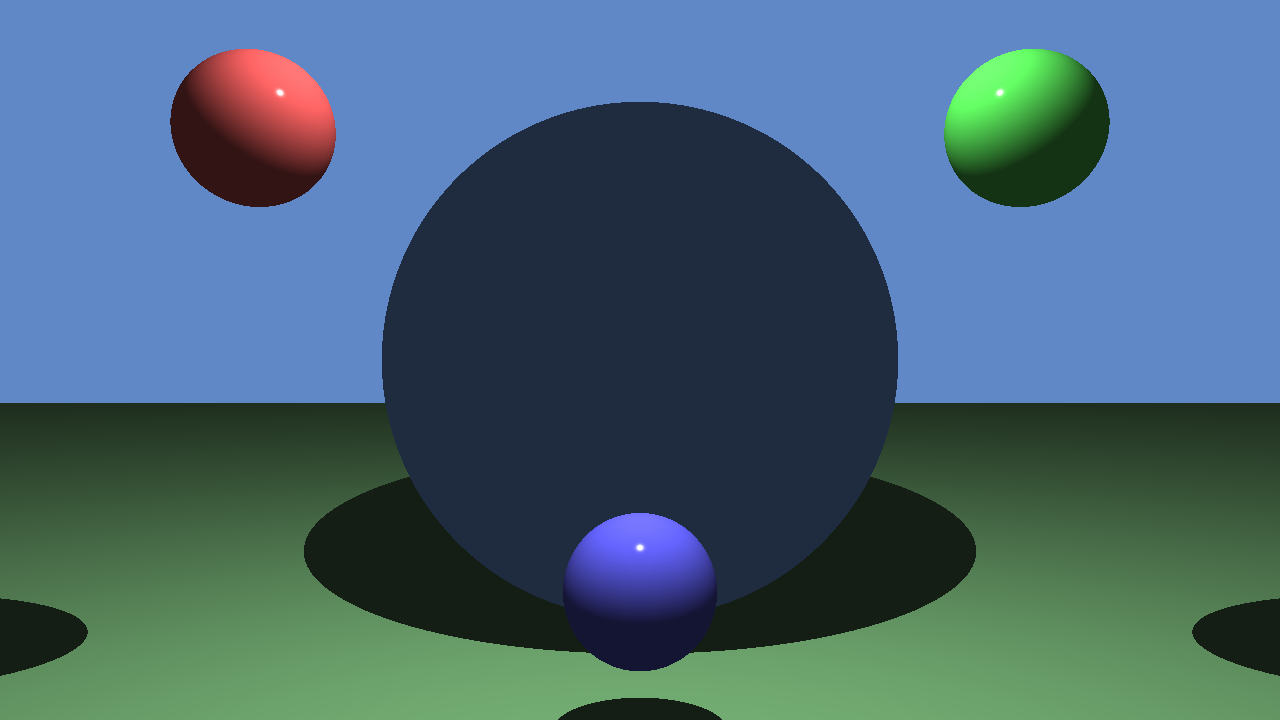
I’ve reverted the project back to before refraction and reflection got added. I will first try to implement reflection and refraction will be implemented after that.

**12:30** Create pseudocode for reflection

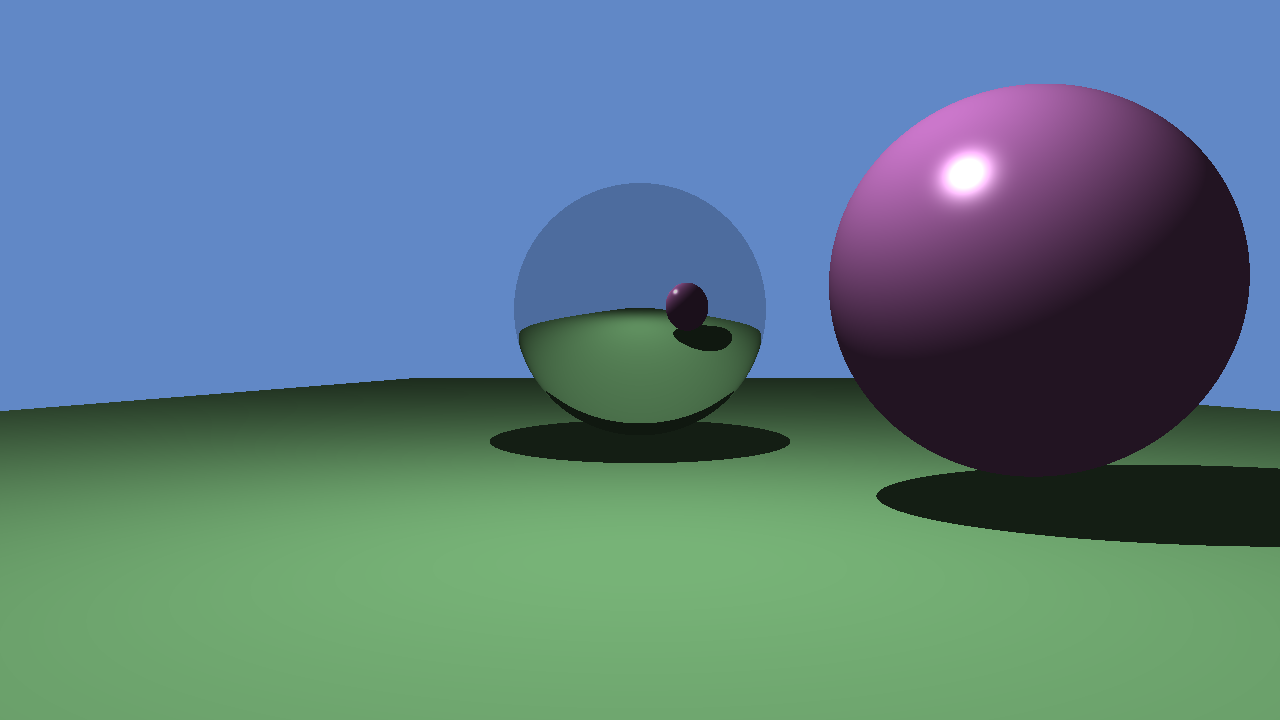
// Inside of getcolor   
  
surfaceColor = intersectSurface.color  
if surface is reflective  
 surfaceColor = reflectionColor  
 Calculate specular ray  
 B = Dot the normal and the incidence direction  
 reflection = incidence direction - 2\*B\*normal  
 if intersect(reflection)  
 return getcolor(reflection,depth++) \* 0.8f  
 else  
 return backgroundColor  
  
for every light  
 if not in shadow  
 color += lamb(surfaceColor) + blinn(light)  
 color += surfaceColor \* ambientIntensity

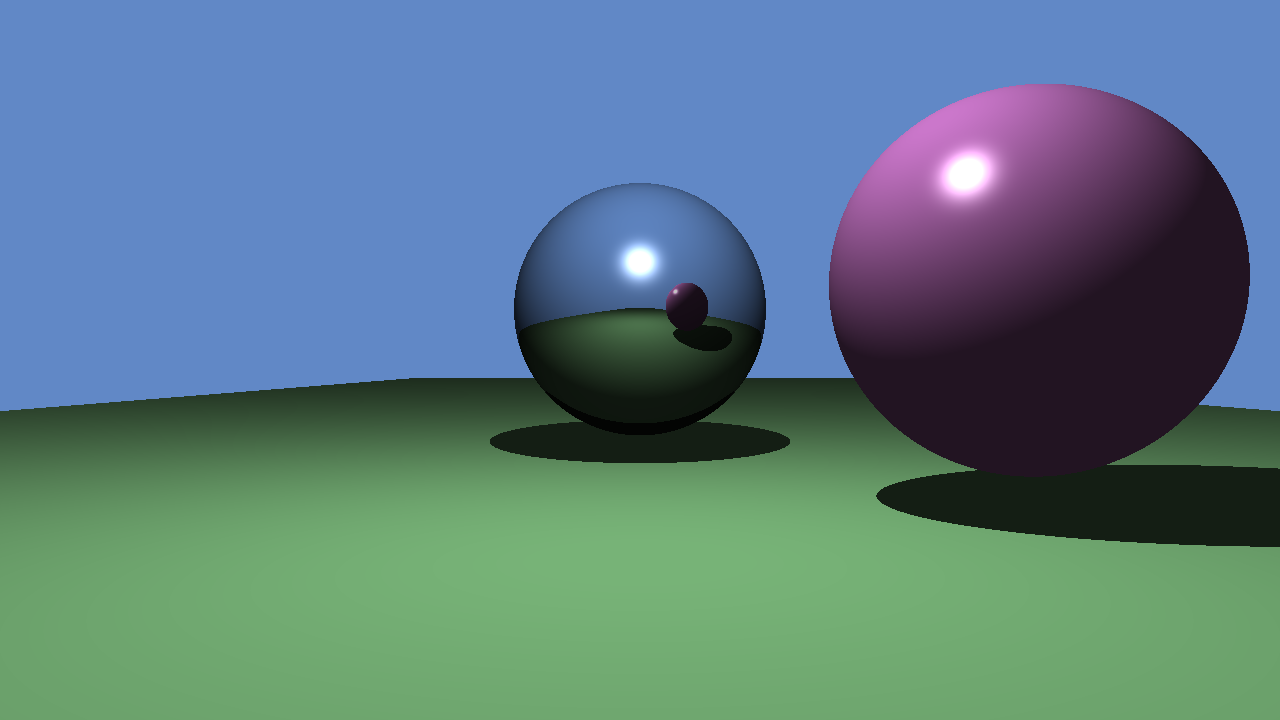
**13:20** **Help out peers**

**14:40** Implement reflection

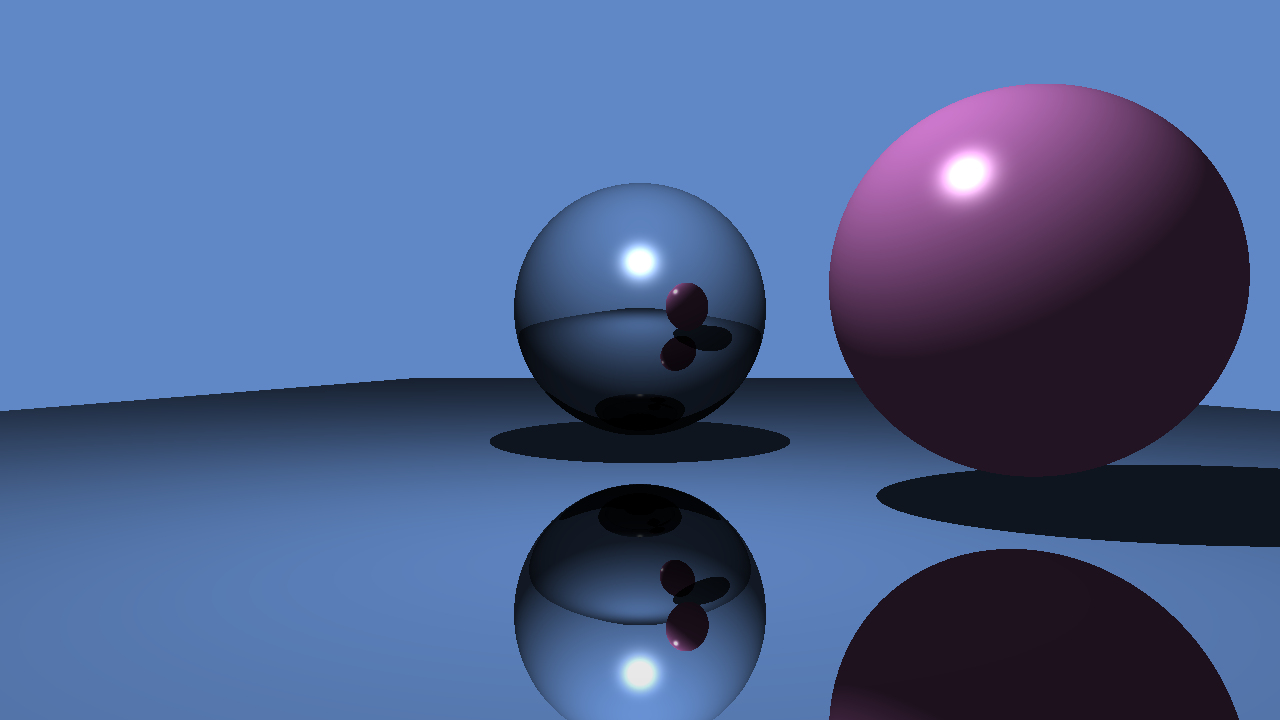


Getting the background color



Reflecting the colors of other surfaces

Adding lambertian and blinn-phong to the reflective surface

Tested support of multiple reflective surfaces

# 

# 

# 

## 

## 16:30-17:15,18:55-19:15 Read notes refraction and study scratch a pixel

**16:30** Researching own notes. I still struggle a bit with the unit circle

**17:00** I finally see that you are supposed to look at this like triangles. The reason why c is cos(o1)\*N is because c is adjacent and the hypotenuse is 1 in a unit circle. So in this case, C is a vector that is equal to the direction of N and the length of cos(o1). Applying this logic to all the other parts clears things up for me.

**18:55** Continue reading the scratchapixel article.

**19:15** Having read my own notes and scratchapixels explanation, I don’t understand why the dot product between the incidence ray and surface normal should be positive.

# Tuesday

## 13:50-14:15 Fill out today’s work log

## 

## 14:15-14:30 Research the applications and definition of the dot product

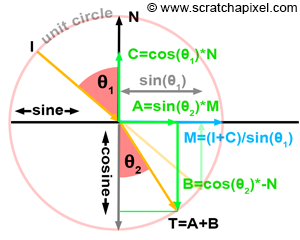
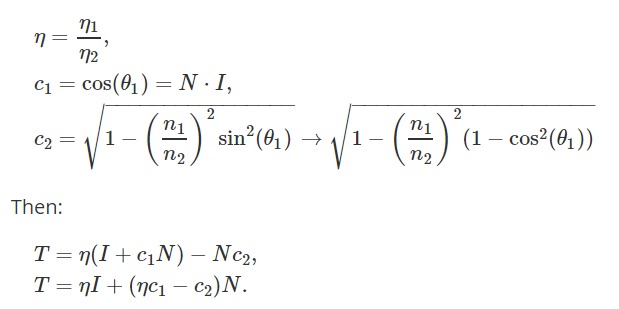
**14:15** Read [Scratchapixel’s](http://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/math-operations-on-points-and-vectors) article on it

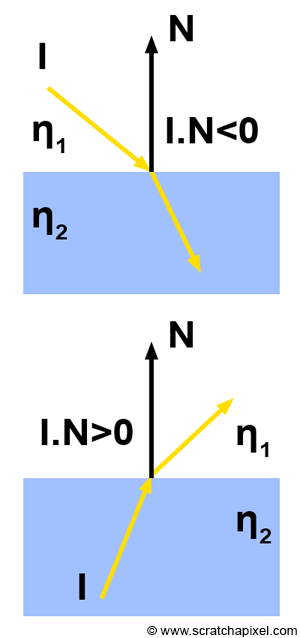
The dot product/scalar product requires two vectors and can be seen as an projection of one vector onto the other. It returns a scalar.

The formula of the dot product is A\*B = ||A||\*||B||\*cos(o). If both vectors are normalized,

A\*B = cos(o). This means that the arc cosine of the dot product returns the subtended angle between the vectors.

## 14:30-15:15 Research about the dot product in refraction



If **the value in** **c2 where we want to take the root square of is negative** and the ior of the new medium is lower than the ior in the current medium (**n is larger than one**), a phenomenon called **total internal reflection** **will happen**. This means that the ray will be reflected, not refracted thus it will not leave the medium. Total internal reflection occurs when the angle of the ray of incidence between the surface normal is larger than the critical angle.

I understand what the article means with reversing the normal. The formula uses the normal to calculate the new ray. In the formula the normal faces in the same direction as the incidence ray. As you can see, when the dot product is negative the incidence ray enters the surface. When the dot product is positive however, the normal faces the wrong direction for the formula and has to be reversed.

Finally the dot product always has to be positive. This is probably because the formula uses the dot product to calculate a projection of the normalized incidence ray onto the normal. The direction would be reversed if the dot product would be negative.

## 15:15-16:00,16:15-17:30, 19:40-20:30 Implement refraction

**15:15** Plan implementation

Like reflection, refraction will happen in the getColor function. The getColor function is actually getting quite incorrect. What it is doing right now, is checking certain conditions of the surface that is being hit. Depending on some factors of that surface, it calls the correct method of calculating the pixel color. This can be casting a reflection ray or casting a refracted ray. I already know that I will need 2 to add two color values for surfaces that reflect and refract rays. Because of this. I will probably add a function called handleIntersection. This function will check the intersection data and decide what needs to happen after that. It may call getSurfaceColor and calculate the color. It may also calculate a reflection ray and calculate continue with that or it may calculate a refraction ray. Its responsibility would be to calculate the correct intersection data used for the pixel color.

We should probably call it calculatePixelColor. That seems like a more appropriate name. calculatePixelColor takes a ray and a scene and calculates a color based on that information. calculatePixelColor will use functions such as calculateIntersection (responsible for calculating intersections with rays and surfaces, it gives intersection data back). Then with that intersection data. It will check what kind of intersection is happening. If it is reflective, it will call calculatePixelColor with that new ray and continue until a suitable color is found. If the surface is refractive, it will call itself with the refractive ray.

calculatePixelColor is responsible for getting the final pixel color and it does so by casting the correct rays. This will be how I will change the ray tracer model

**15:30** Add refraction index to JSON file

**16:15** Refactor project to support refraction

I’m not sure how to get the old refraction. There are a few designs that I could use.

* I could assume that the user never has two mediums that have a different iors. If this is the case. I can assume that the ray is always in the air or in any other medium. We can use a constant ior of 1 for air and check if the ray is currently inside of the medium or not.
* I can add a member variable to the ray tracer that keeps track of the current ior. I’m not sure if the ray tracer should know this. It is definitely something that no other function needs but calculateRefractionRay.
* I can keep track of the old ior inside of the getPixelColor function. This solves both of the above problems but it makes the function responsible for more things which goes against the c++ core guidelines [F.2](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#f2-a-function-should-perform-a-single-logical-operation).

I think that the last solution is the best one. It doesn’t make the function that more complicated. I’m not sure if a function called calculatePixelColor should be responsible for something like that. That means that the function name is not that clear.

**17:30** Refactoring complete

**19:40** Implement the refraction ray function to calculate refraction ray

**20:30** Continue implementing refraction

# Wednesday

## 16:10-16:20 Fill out yesterday’s and today’s work log

## 16:20-17:50,18:45-20:40 Implement formula to create a refracted ray

**16:25** I just realized. How am I going to know the refraction index when I leave a medium. When the ray tracer calls the createRefractionRay function for the second time on the same surface (because it collides with the edge of the current medium). It will use the same refraction index. How is it supposed to know that the new refraction index should be the air or another surface?

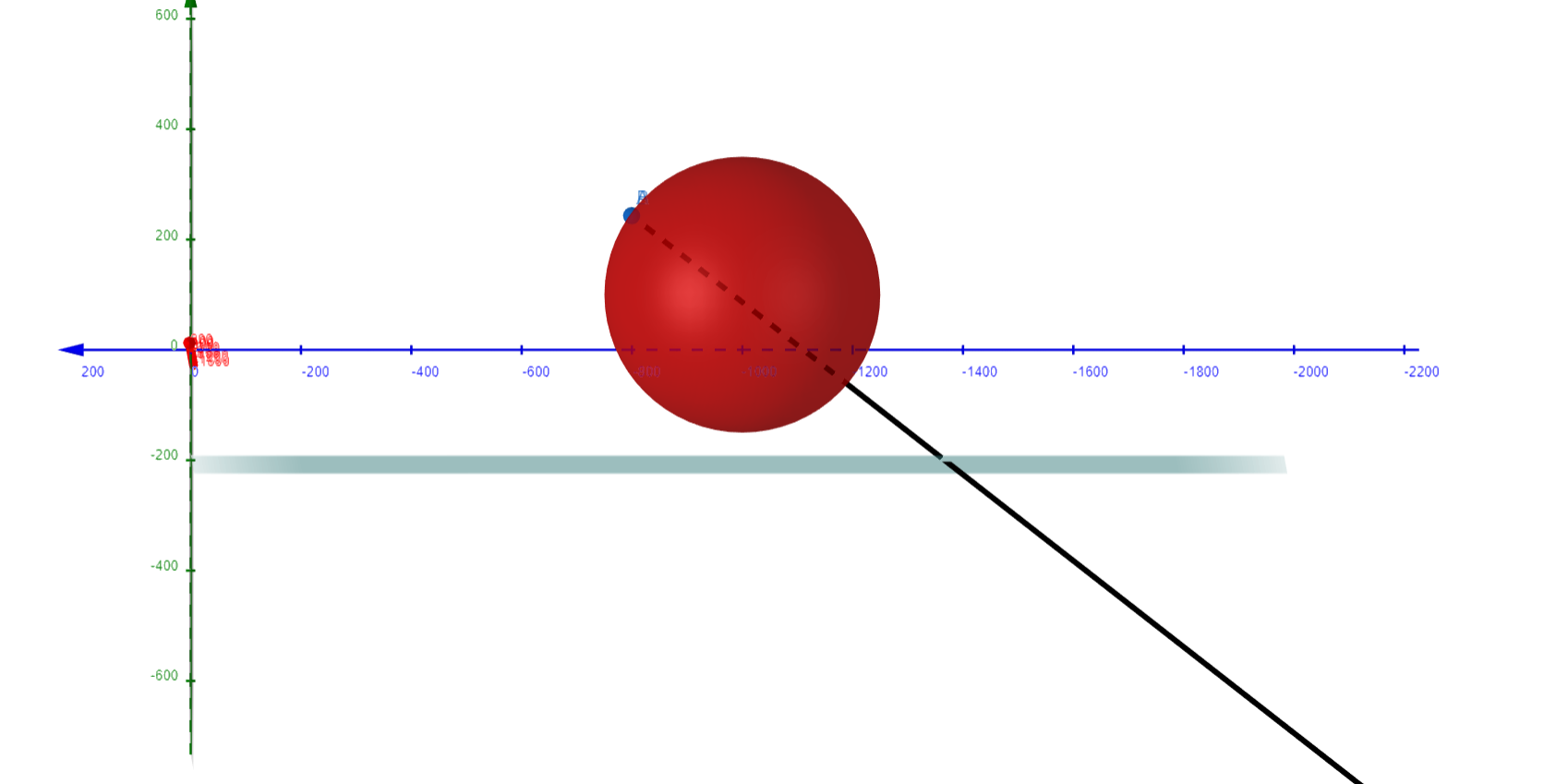
Well, that is actually not possible. You can’t have two refractive surfaces inside of each other. One can intersect the other but they can’t live inside of each other at the same time.

For now, I’m not gonna implement multiple refractive surfaces inside of each other because the reference image doesn’t need it. But it is something that I have to keep in mind

**16:45** Continue implementing refraction ray generation

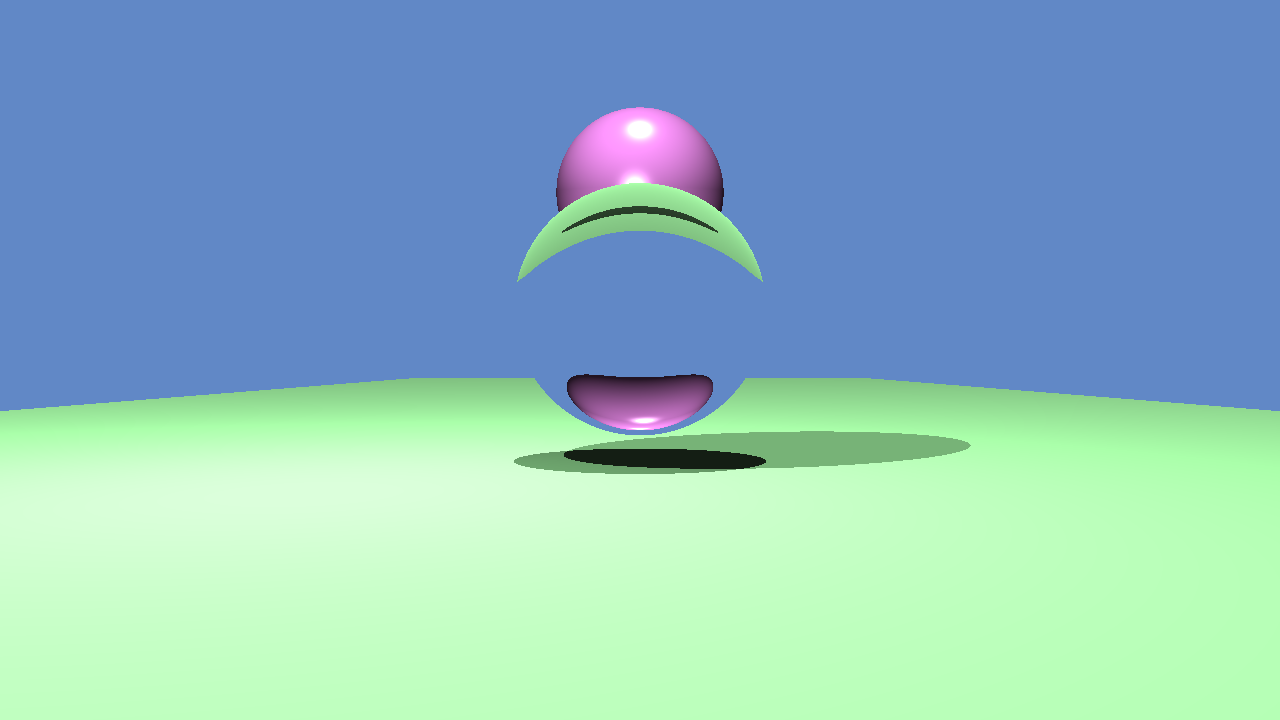
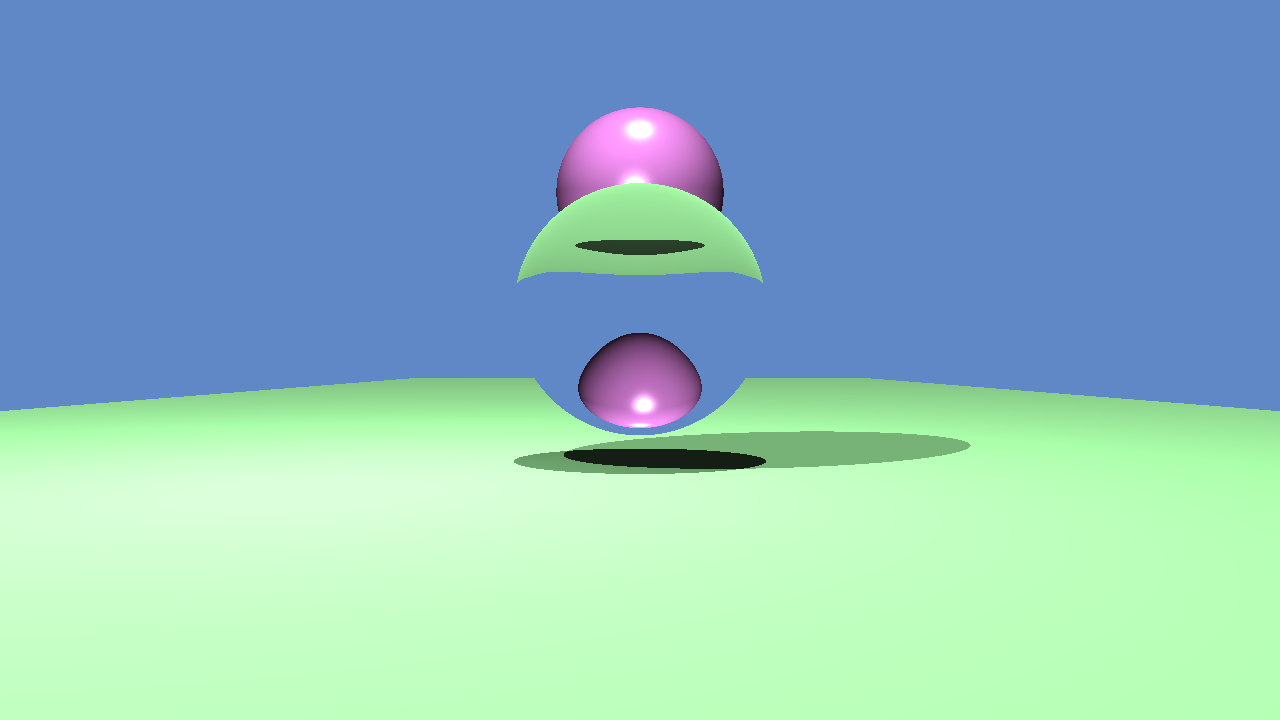
**17:00** Test our refraction

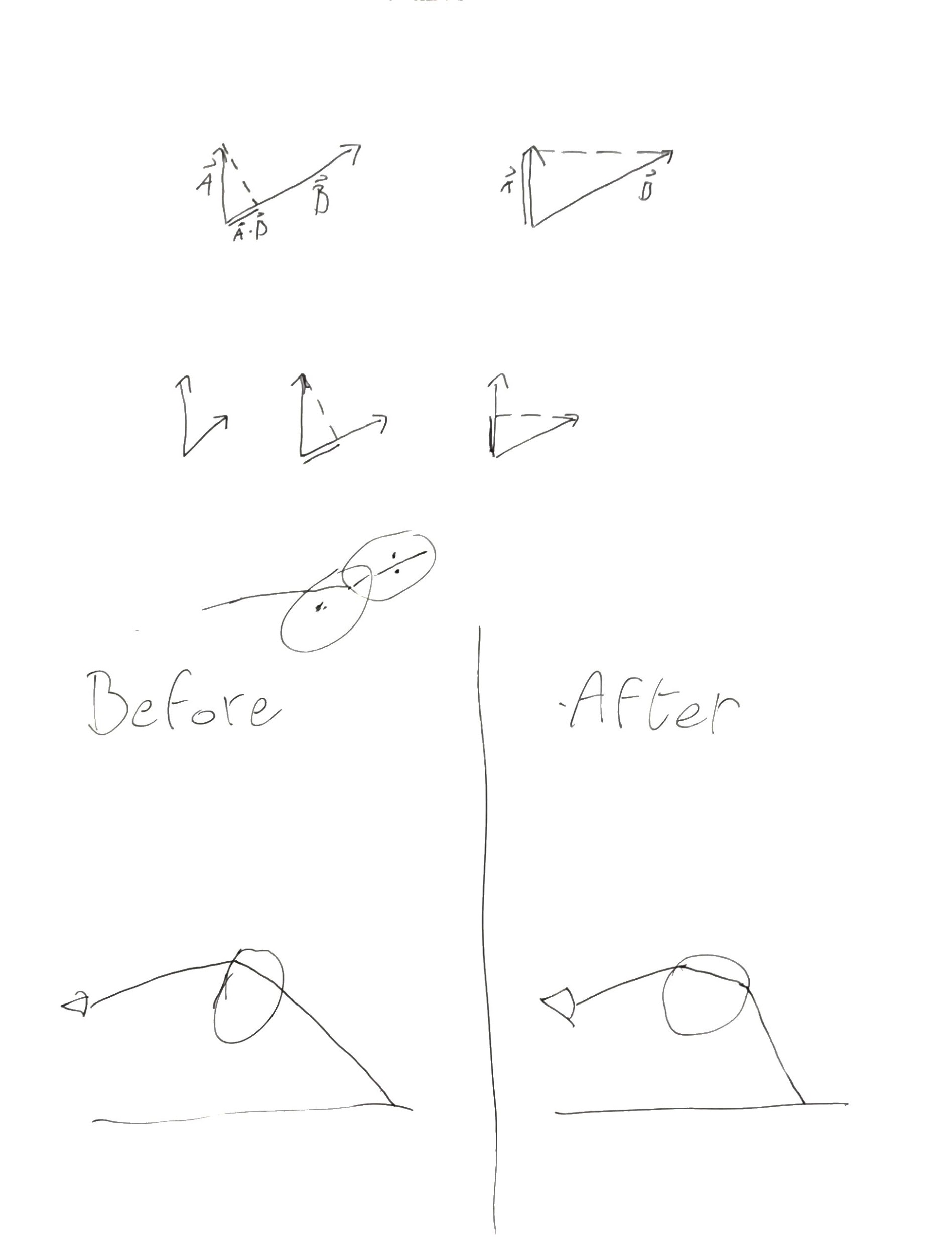
**17:30** Recreate scene in a plotter

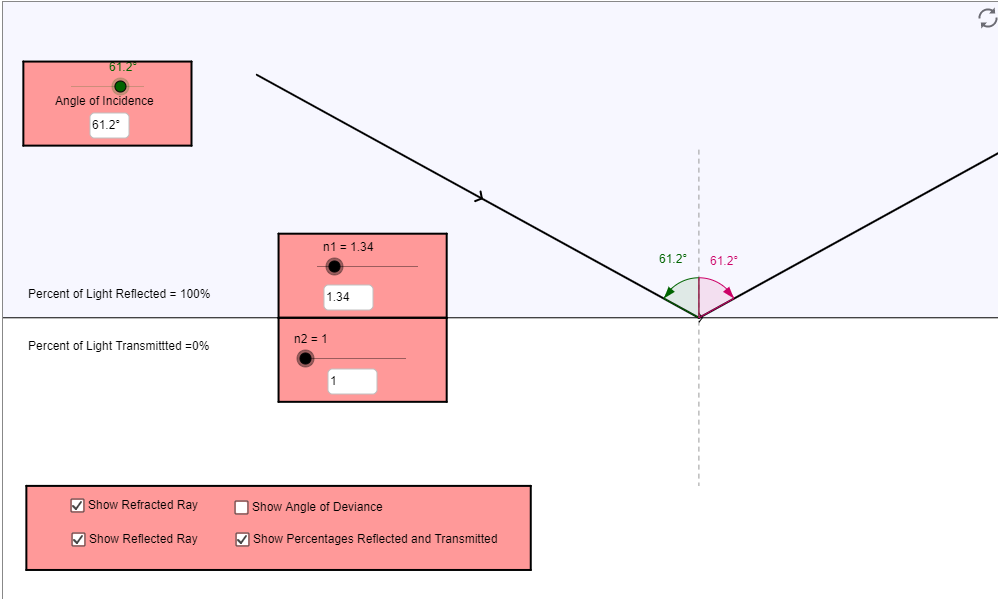


There should be another collision with the sphere, but it goes straight to the plane.

**18:45** Continue debugging the error

**19:10** I found the mistake. In the sphere intersection function, I take the smallest multiplier of the two. Because I never thought that I would cast rays inside of a sphere, I just used the smallest of the two. This means that one of the two could be negative, meaning that the returned value also is negative. Because I check if the final value is negative, the whole sphere intersection got skipped. The refraction went from this to this.[](https://drive.google.com/open?id=1eq4t2DMonZTcIsKkJ-sbNC1mlLgOUi3n)[](https://drive.google.com/open?id=1FwKGFO4VQkyvrSOL9bv-PmkBOOj6kUuQ)





**19:20** Implement total reflection

Total reflection happens because the refracted angle is larger than 90 degrees. This means that the light ray never leaves the old medium. [This video](https://www.khanacademy.org/science/physics/geometric-optics/reflection-refraction/v/total-internal-reflection) made it very clear.

**20:00** I found the error

Again a very stupid error on my part, but there is a variable in my formula called c2. That variable uses the refractionRatio. The refractRatio gets inversed if the refraction happens inside of the object. I calculated c2 before that.

## 20:40-21:40 Test refraction

I’ve been busy with this for an hour now and I can’t create a situation where the final image looks different with and without total reflection. I’m not too sure how to make such a scene. I know that total reflection is more likely to occur when the angle between the normal and the incidence ray is larger, but even with a very small sphere, I can’t produce anything. I will probably face this problem when I have to recreate reference image 1 and I will fix it then.

# Thursday

## 7:00-7:10 Fill out today’s work log

## 7:30-8:20 Research about the Fresnel equation

**7:30** Research definition and article

The amount of transmitted light increases when the angle of incidence decreases.

The Fresnel equation describes how much light is reflected by one of the two perpendicular waves light is composed of. The details of these go beyond the scope of the tutorial. By taking the average of the two Fresnel equations, we can calculate the ratio of the reflected light ray.

The formula for both of the Fresnel equations are:

F∥ =(η2 \* cosθ1 − η1 \* cosθ2 / η2 \* cosθ1 + η1 \* cosθ2)2

F⊥ =(η1 \* cosθ2 − η2 \* cosθ1 / η1 \* cosθ2 + η2 \* cosθ1)2

This will return a ratio meaning a value between 0 and 1. To get the ratio of reflected light, we take the average of these two formulas.

Fr = (F∥ + F⊥ )\*0.5

Of course, if sin()2 is larger than 1. We have total internal reflection. We can then return 1 for the reflected ratio. Though I’m a bit confused because total internal reflection should only happen if the ior of the new medium is smaller than the old one.

**18:00** Realization that sin()2 can never be larger than 1

Basically, when sin()2 is larger than 1, it means that the new angle is larger than 90 degrees. Is is not possible to have a value of sin larger than 1, because than the ray would just go back inside the old medium. This means that it is impossible for sin()2 to be larger than 1 if the medium the ray is entering has a smaller ior than the previous medium thus the check for the ior is redundant.

## 8:20-8:50, 9:15-11:30, 12:20-16:30,18:00-18:40 Implement refraction

There are some changes that have to happen to the ray tracer:

* Objects can’t only refract light. They always reflect light. The ratio of this is decided by the relative iors. I have to change the format of the JSON data files to support this and the getPixelColor function to take this property into consideration.
* There is a redundant check in the calculation for a refracted ray. We can check if the sin()2 is larger than one. This is also a comparison that I can explain instead of the c2 < -. Though it is obvious that you can’t take the square root of a negative value, I’m not sure what variables decide the value of c2.

Let’s start with the second one.

**8:25** Change the way refracted rays are generated

Snell’s law is as follows:

n1/sin()2=n2/sin()1

This means that sin()2=n1\*sin()1/n2

sin()1 = n1\*sqrt(1-cos2()1)/n2

cos()1 = dot(incidence,normal)

**9:15** I noticed no difference in the refraction calculation

**9:50** Change format of JSON files and design of ray tracer

I assume that all surfaces that are transparent also reflect some light. That means that we have 3 different surface types: normal, reflective, reflective and refractive. To specify these properties, surfaces a float called **ior** and a float called **surfaceColorRatio**.

All variables are between 0 and 1. Ior is self explanatory. surfaceColorRatio is the ratio between how much surfaceColor is applied and how much reflection/refraction color is applied. This is because water and glass can have colors. I can’t confirm that this is the correct way but I think that it is good enough to simulate the reference images. To know which function the ray tracer has to call it does the following:

// In calculatePixelColor

Vec3 surfacePixelColor

Vec3 reflectPixelColor

Vec3 refractPixelColor

// This means that part of the color should be reflection or refraction

If surfaceColorRatio is not 1

ReflectionFresnel = 1

// If ior is different, so is refraction

If ior is not 1

ReflectionFresnel = getFresnel(data)

if(ReflectionFresnel is not 1)

refractPixelColor = calculatePixelColor(refract(ray)) \* (1.f - ReflectionFresnel)

reflectPixelColor = calculatePixelColor(reflect(ray)) \* ReflectionFresnel \* 0.8f

Return surfacePixelColor \* data.surfaceColorRatio +

(reflectPixelColor + refractPixelColor) \* (1.f - data.surfaceColorRatio)

This should do the job. I’ll be implementing it now

**10:20** Implement function to calculate Fresnel

I can’t really test it without implementing the new design for calculatePixelColor function. So that is the next step

**11:20** Implement new design for calculatePixelColor function

**12:20** Add variables to the JSON file

I’m going to use templates to check if my keys exist. This should save me a few lines of if else statements.

**13:00** Discussions

**16:10** Continue work on the ray tracer

**18:00** Continue work on calculatePixelColor function

In the end I used a way simpler model. Instead of trying to find these clever paths, I divide surfaces in three types: diffuse, reflective and reflective/refractive. This allowed me to easily do the calculations as follow

If not over max depth and intersection happened

// If surface is diffuse

if surfaceColorRatio == 1

color = diffuseColor

// If surface is reflective

else if ior == 1.f)

color = diffuseColor \* surfaceColorRatio +

reflectColor \* (1 - surfaceColorRatio)

// If surface is reflective and refractive

else

reflectionFresnel == calculateReflectionFresnel

if reflectionFresnel != 1.f)

color += refractionColor \* (1 - reflectionFresnel)

color += reflectColor \* reflectionFresnel

else

color = background color

return color

**18:15** Testing out solution

## 19:05-19:45 Research procedural texturing

Procedural texturing is the process of texturing a surface with mathematical operations. To apply texturing, you need texture coordinates. Texture coordinates are object vertices mapped to 2d space or texture space.

Most of the article is about how to make patterns. That was the title. I will first attempt to recreate basic texturing for my surfaces.

The process appears to be as follows. When we get the color of a surface, we call a function. This function maps the point to texture space. It plugs the texture space coordinates inside a mathematical function and returns the color.

I need to find a way to convert to texture space.

Learn about [polygon meshes](https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh)

The article referred to this before diving into it. I’ll research that now

# Friday

## 13:00-13:05 Fill out today’s work log

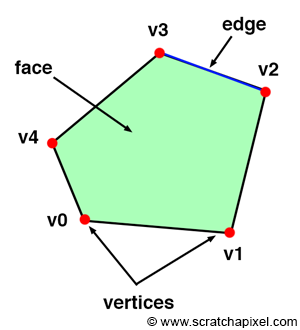
## 13:05-14:00 Mess with google sheets formulas

## 15:45-16:45, 17:15-18:15 Research article about polygon meshes

**15:45** Research the correct approach to researching this

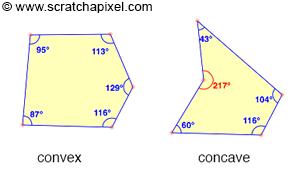
Before I started to read the article, I decided to research it a bit more. Scratchapixel has been updated for the last time in 2016 (I think, it says in the footer). While it has been and still is a very reliable source I do have to broaden my sources a bit. This is because I’m not sure if I’m being efficient with learning from scratchapixel.

After reading a bit I have decided that I will stay at scratchapixel for the time being. The book Fund. of CG is quite abstract and goes more in the theory than the explanation. This is not a bad thing, but sometimes I can’t grasp how they came up with a certain formula. Scratchapixel almost always leaves an explanation for those things. I feel more efficient learning from scratchapixel. If I can’t find something there, I will probably search up other sources, But I’ll stick with scratchapixel for now. Fund. of CG also has a whole chapter about texture mapping, I can always use that if I can’t find my answers here.



**16:05** Read article about polygon meshes

**Polygon meshes** (aka meshes) are one of the (in not the) oldest form of geometry representation used in computer graphics. A **polygon or face is a shape which is defined by connecting it’s 3D vertices.**

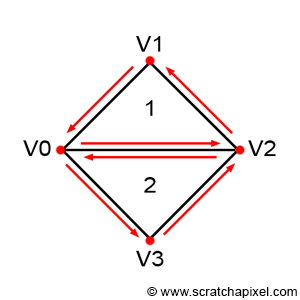
A vertex is a point. The face is defined by connecting these point (counter-)clockwise (also known as winding). The line connecting two vertices is called an **edge**. A face can have a minimum of 3 vertices. This is then a **triangle**. If you have four vertices it is called a **quad**, and if you have more than four vertices, then it is simply a **general polygon**. 

To make a face, all the vertices need to lie in the same plane (generally).

**Connectivity** is the information on how vertices are connected

**17:30**

The notes till now are a bit obvious, I’ll be more specific now. We need to know the following things about polygons:

* The number of faces/polygons that the object consists of. A cube has 6.
* The number of vertices each face needs. An cube need 4 per face (quad).
* The order of vertices for every face. Look at the figure on the right. It consists of 2 faces, every face needs 3 vertices. For the order of vertices I’ll go in a counterclockwise order. This would look like this: 0,2,1,0,3,2  
  By using the information of point 2, we know how to interpret this array. The size of this array is equal to the sum of all integers in the previous array that specifies the amount of vertices a face needs.
* The positions of the vertices.

All the data is located in arrays. In the same order, a cube would look like this:

* Number of faces: 6
* Face index array: [4,4,4,4,4,4]
* Vertex index array: [0,1,2,3,1,5,6,2… (goes on till 24 (6\*4))
* Vertex array: [[-1,-1,-1],[1,-1,-1],[1,1,-1],[-1,1,-1[... (goes on till 8)

## 17:45-18:20 Research different data formats than JSON

I got distracted and started reading arguments about JSON vs YAML. While creating scenes, JSON got a real pain to use. When I was prototyping scenes and I wanted to remove one object temporarily, I usually just comment it out, but comments are not allowed in JSON. JSON is very human readable, but I do not find it that human editable. YAML does allow comments. It also has a lot of functions that I don’t need, but may be useful in the future. The main argument for me is that it allows comments which helps me improve workflow.

I was also looking at TOML which seemed very nice and simple, but TOML can get a bit complicated with nested arrays. I have quite a lot of those. So that is why I will use YAML soon.

## 18:50-20:30 Research polygon meshes

The process of applying an image onto a surface is called texture mapping. The process where faces are being laid out on the texture is called uv layout. The shape on the texture represents the shape of the face. If this is not the case, the texture would most likely appear stretched.

# Saturday

## 16:45-17:25 Read about texture space

You specify all the information yourself. You define the vertices for the polygon. The way to store this information is noted in yesterday's notes. The two arrays vertex and texture positions are indexed by the vertex index array. You could split these two.

You specify the texture positions in a 2D space and they correlate to their vertex position counterpart.

I'm not too sure on how to calculate the right position based on a hitpoint in the 3D space. I'll have to do research on that and it wouldn't hurt to look at standard formats.

For some reason RenderMan sees vertex position oas an optional parameter. That is also for the normals. I'm not too sure why it is an optional (the position) and why you would save normals. Maybe you can lookup the normal by comparing the 3 closest vertices with the face index array. I'll research it a bit more.

At least I won't be needing to define any meshes. I only need to define vertex positions. It will be hard texturing a sphere because we need to map planes on it to a 2d space. Planes shouldn't be too hard to procedural texturize. I just need a way to map between vertex and texture positions.

## 17:25-17:40 Think up a way to do texture mapping

Scratchapixel defines the process as taking the face and pasting it on a 2D surface (the texture). A plane is already ready for this. You can define a plane with 4 vertices. These vertices are in 3D space. We need to be able to convert them into 2D space. I think that it is possible by rotating the plane to be co-planes with a plane which edges consist of the axises (that sentence came out needlessly complex). By rotating the surface and hitpoint, the surface is mapped to 2D. Then the hitpoint is just located somewhere in the 2D space.

## 20:15-20:30 Recreate first reference image

I'll have to touch a bit on refractions because it looks quite different. My refraction also ignores blinn phong.

## 21:45-22:15 Devise method to texture map planes

I think that I know a method to implement texturing. If I know the vertex positions and the hitpoint, I can use linear interpolation on the vertices, though I'm not sure how this would work in 3D, only in 1D.

I'll read about interpolation methods first then.

# Sunday

## 19:00-19:45 Create method to texture map plane

I first need to change the way we get the surface color of a surface. I will make this a function (should’ve made getters in the first place) that takes the hit point.

It is actually quite hard to implement this function. It is dependent upon the surface. For example, in a plane, we use the width, height and depth because we interpolate between those coords. But we can’t define functions inside of the scene file. We also can’t hardcode procedural texturing in the class.

I first wanted to make a structure that contains function pointers. In the scene file, you can specify the index of a function pointer if needed and the getColor function inside of the material would call that function. This unfortunately is not possible because a surface knows nothing about the position and size of the surface.

One way to fix this is by using vertices only. Every surface has 3 or more vertices. We can use a vertex index array and see where the hitpoint is closest to. Then we interpolate between those 3 points. But this sounds needlessly complicated.

Right now I’m considering if I should just add a hack to the plane creation formula. Instead of fully implementing textures. I’ll just make getter functions for the surface color. During creation if the flag checkerboard exists and is true, it will pass the function that is supposed to procedural generate. I’ll probably do that and work further from there because texturing is a huge topic and I should focus my attention more on the actual ILOs like optimization and a camera.

**19:15 Designing the hack**

To implement this hack. I will add a virtual function to the surface class. It is called **pointToTextureSpace**. I dislike this design because it bloats the interface but every surface does have a texture space. Whether we use the space is up to the person who design the scene. The IntersectionData class will **contain the uv coordinates** of the hitPoint. The material class will get a **getter for the color** and during the creation of the material class it will be **passed a function pointer**. This function pointer decides what color is returned, resulting in procedural texturing. The only issue that I have is that I have to define more colors or I will define it in the function, probably the latter. I will add a variable to the data file called **texturingIndex**. I will hard code the procedural texturing functions and push these into a static array called **texturingArray** in the material class. This array will be filled with procedural texturing algorithms.

I’m not sure if texturingArray should be a global variable or a static member variable (is that even the right term because static variables in classes can’t call member variables). I’ll research it a bit because I should know the differences between those things.

After researching it a bit it appears that static has local linkage meaning every file will create its own copy of it. I will use a static member variable.

## 20:15-22:00 Implement the hack

I had some oversights. The application class now needs to know about materials because it has to define the procedural texturing functions

**21:20 Error in derived formula**

I have created a simple formula that is supposed to calculate t between two points and the hitpoint. But for some reason this formula is returning negative values. While t is normalized thus should be between 0 and 1.