# 0. Preparation

Created by Anton Hritsan, last modified on Oct 19, 2021

Table of content



## Software

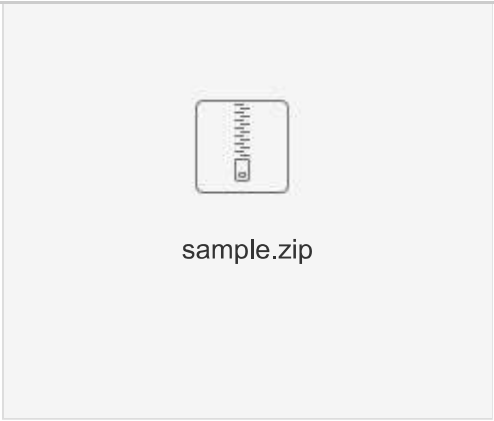
For current training you need to install

1. Visual Studio( any of )
2. Git for Windows
   a. can be installed as part of Visual Studio
3. Tortoise git
4. git lfs
5. python 3.x
   a. [ Optional ] addition part of Visual Studio for debug python scripts
6. Text-based editor.
   a. you can use whatever you want, Notepad++, Visual Studio Code, etc.

All minimal softs you can find in our internal machine \\lviwks0126\pub

## Simple sample

A tiny sample with primitive 'drawing' aka 'backbuffer' by Win32 API you can find in .zip archive below
› Sample you can get here




# Repository

Your trainer will give you a link for your personal repository. it's literally your repository, you can do whatever you want with it. Also, you should get an archive with initial sample.

## Configuring git repository

Before you start, here's some mandatory things to do

1. git clone you repository
2. copy sample to your working copy of repo and launch it sample.
3. install git lfs for your repository
    a. type 'git lfs' in your gir repo folder to see instructions for 'git lfs' usage
    b. you need to install git lfs for your repor
    c. you need to add at least one track file by console
        i. every next, can be added by editing by any text editors file '.gitattributes'
    d. Add all binary files in repo to 'git lfs'( for example *.exe and\or *.pdb )
4. add ignoring files\folder along with adding to git lfs files\extensions
    a. before commiting tour changes, compile your applkication in 'Release x64' scheme!
    b. [ mandatory ] add to ignore all debug folder(s)
    c. [ mandatory ] add to ignore 'delete_pdb' folder
    d. [ mandatory ] add all Visual Studio folder to ignore!( .vs folder for example )

## Processing your tasks

Git repo - complex thing. Master branch - it's a life of your application. It literally means - you have to have a stable master branch.

Please follow next guides during completing your tasks:

1. Master branch has actual and stable solution for your last completed task
2. Master branch has to have stable binary to launch your application. So, anyone with an access to your repository can see your progress without re\de\compilation and fixing bugs\errors
    a. Readme.md must contain info about how and what to modify in your application to check addition tasks. if you have config for your application, put there some kind of trigger to check addition tasks WITHOUT re-compilation. Anyway, need explanation how to check your changes in readme.md
3. For every group of task( including this group ), you have to have a separate branch. In this 'work branch' you can do whatever you want, but remember, history will be in master soon...
4. In the end of the work day, you have to commit\push your changes.
5. After completing tasks for a whole 'group', your work branch HAS TO be merged with master( do not forget about 'Release' build before merging )

As you saw in main page, you will do an own ray tracer (reverse\back ray tracing) . This is not hard and you don't need to know whole math in details. You will use the basics of linear algebra and primitive things from analytic geometry. Actually, all necessary knowledge from 8 and 9 forms from general school programs. This ray tracing will not use any 3rd party technologies or SDKs, you will make a ray tracer only with power of CPU and basic math rules. During training, you will face interesting challenges to implement or re-invert.

> [ ETA ]This training is estimated to 2 (two) work weeks, or up to 10 work days. Prolonging is possible as exception and such possibility will be discussed in a personal way.
>
> [ Code ] You're doing your own ray tracing. You are not pushed to use any company's code conversions and\or common practices. Feel absolutely free, it's your own personal project now.
>
> [ Misc ] You can use anything in this world to get this training completed. But please, think twice before copying info from internet. People around you in the office can help you to find right direction for your particular task, but they will not provide you a complete solution.
>
> [ Samples code ] In all documentation, you will see samples of code. you will not have any chance to copy it, they all will be attached as a picture. I'm sure you're good at typing, but I want to be sure you understand what you're doing.

## Math in training

In sample application you have implemented basic classes for ray tracing. With time, you will add functional to them.

What do you need to make ray tracer.

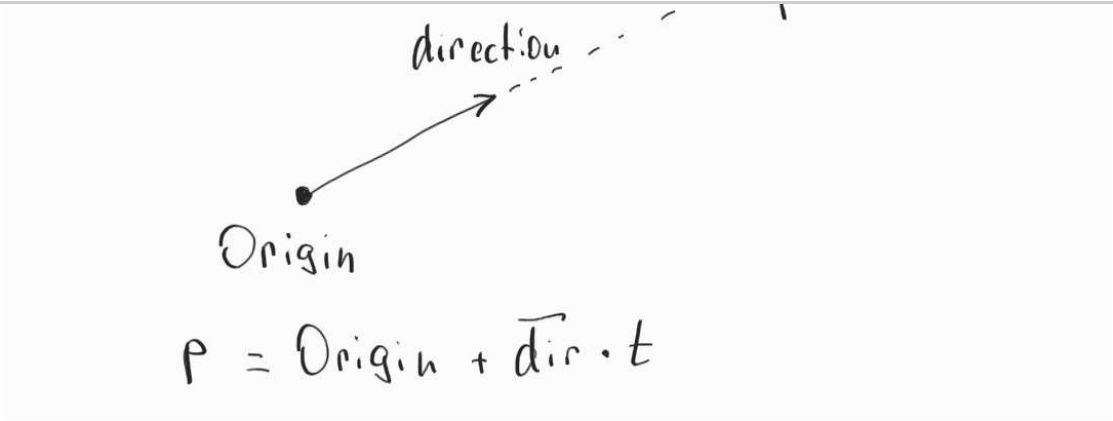
1. Vector in 3d
2. Ray
3. Color representation
4. ability to use vectors, colors and rays

### Vector

Vector is a a structure that hold 3 floating numbers. We can use these numbers to describe several things at once. this can be direction, position, as far as color description. In sample, you can find such class as 'Math::float3'.

### Ray

Ray is a struct that holds position and direction. Please note, direction in Ray can be not normalized due to the fact it is not necessary. The basic thing of ray, you can find any possible point of this ray, the only thing you need to get is the distance from original point.  Illustration of this below (t = distance )



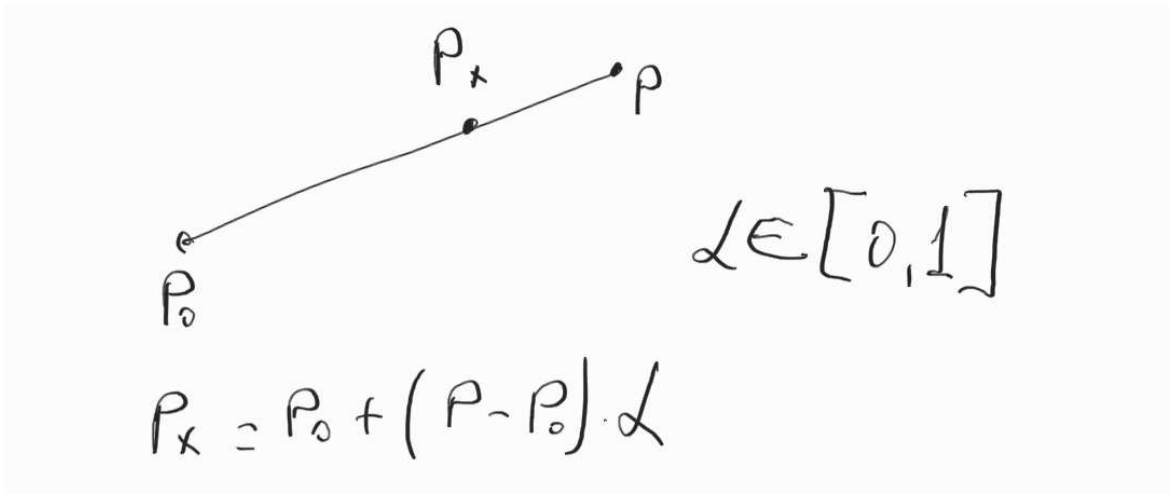

## Color

Usually, color is represented by four components RGBA with some number of bits per channel. For this training, we will not deal with transparency, so RGB will be more than enough for us. As we already have float3 class with three components, we will use it to describe color as well. float3 variables 'xyz' will be as 'rgb' for us.

One more thing about color. As far as we use float for describing color channel, a good idea will be getting one channel in range of [ 0.0f ... 1.0f ] to have maximum precise. For final converting to RGB, you just need to multiply each channel by 256.0f.
NOTE: keep in mind that color channels can not have negative

## Interpolation

For dealing with values in ranges, we might need to know values somewhere in between in range. For our simplicity, we can use only one type of interpolation - linear interpolation. It isn't hard to invent this formula geometrically, in the end you will get something like




Add it in Math base namespace for future usage. Sample code below

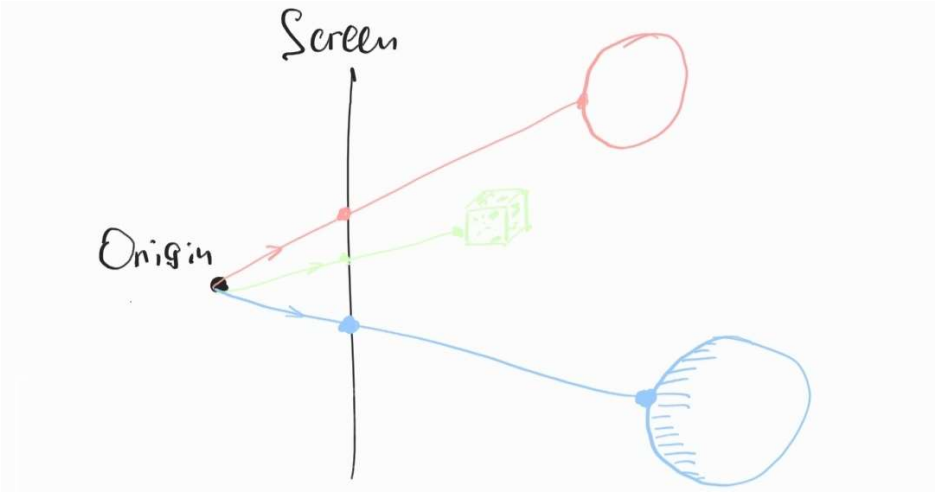
```
    }


    template< typename type >
    inline type Lerp( const type& lValue, const type& rValue, float t )
    {
        t = Clamp( t, 0.0f, 1.0f );
        return lValue + ( rValue - lValue ) * t;
    }
}
```

# Camera and Rays

## General info

For ray tracing, we need to know, the origin point for all rays. As far as reverse\back ray tracing is a a process of spawning ray and storage colors from all intersections, we need this 'origin'. Color that we get from intersection should be calculated from material on surface. Material is a structured description of physic world surfaces( microfaceting, roughness, glossiness and others ).

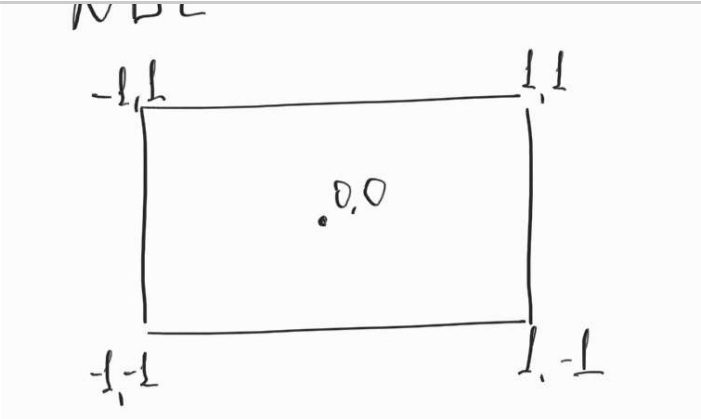
All the process of future work can be described by next illustration



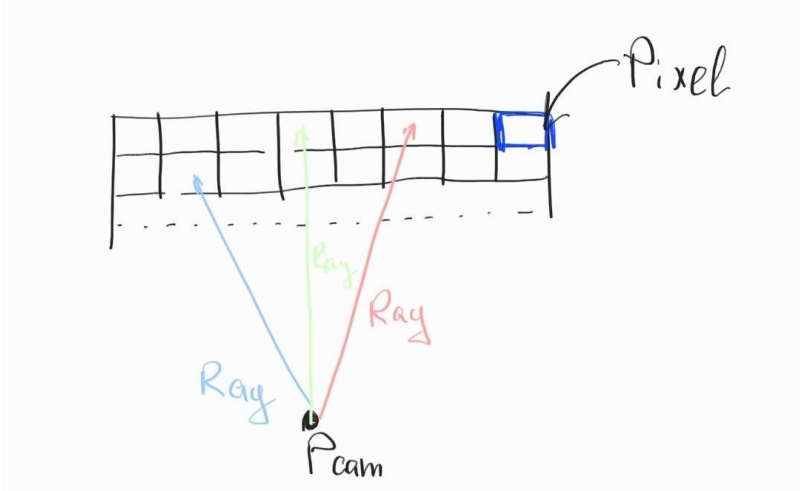

## NDC

we are throwing ray in different directions from our Origin point with saving intersection color from intersection with objects. The logical questions are: What is 'Origin' point? How to generate rays in 'right' directions? What's the amount of rays we need?

Let's examine this idea from different point of view. We have physical limitation of our screen( window ), so we can say, we need at least one ray for each pixel in the screen( window ). Next, 'Origin' point can be described as eyes. So, it's easy to convert it to 'camera'. Only one question remains - directions. To answer it, let's think about our screen. we have limited width and height what doesn't describe anything in 3d world, so we need to deal with limitation in graphics form. Any screens\windows can be treated as NDC( Normalized Device Coordinates ). Preview below in illustration




if we use NDC with screen pixels, we can make rays with directions for every pixel. So, idea is next:




## Viewport Aspect Ratio and Field Of View

Our screen in not  square, so NDC coordinates will not be linear too. So, let's define Aspect ratio first.  Aspect ratio is equal to width divided by height, easy, huh?

Field of view is the angle between opposite planes in frustum pyramid. For example, vertical fov  is the angle between top plane of frustum and bottom one

PICTURE WILL BE HERE!( picture about frustum view )

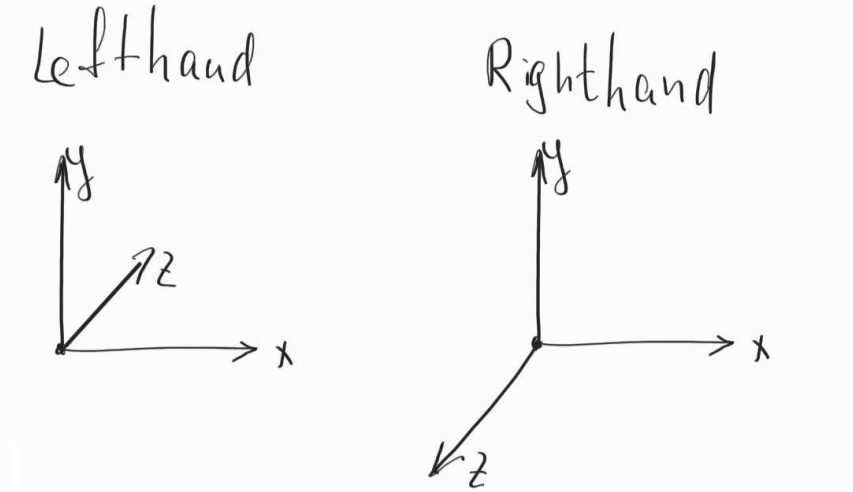
Since we have width > height, we should vertical fov as base one, and horizontal can be calculated as  aspect_ratio * vertical_value. For given NDC with vertical range [ 1.0 ... -1.0 ], we need to put our desired angle of view. More or less common default in gamedev is 45 degree. Next step is calculation of our limits in viewport.

```
float fHalffov = fov * 0.5f;

float cosmin = cosf( Math::AngleToRadian( fMiddleAngle - fHalffov ) );
float cosmax = cosf( Math::AngleToRadian( fMiddleAngle + fHalffov ) );

float3 hor_start( cosmax * aspect_ratio, 0.0f, 0.0f );
float3 hor_End( cosmin * aspect_ratio, 0.0f, 0.0f );

float3 ver_start( 0.0f, cosmin, 0.0f );
float3 ver_End( 0.0f, cosmax, 0.0f );
```

Note: hor_start and hor_end with inverted cosmax\cosmin due to other NDC range( see illustrations above ).

### World coordinate system

Last thing need to discuss before making ray tracer - our world coordinate system. In games, developers use two major systems: Lefthand and Righhand coordinate systems. Mathematicallyit is better to use righthand, but it's not fun of thinking in reverse direction, so I will use Lefthand coordinate system. Illustration with comparing these two systems




Please double note: all samples in internet are with Righthand coordinate system. If you get the idea how it works, you will not care about it at all.

To describe and compare left and right hand systems, imagine your eyes as origin point and you looking forward. With lefthand coordinate system - your eyes are looking in to positive depth, with righthand - your eyes are looking into negative depth( to your back ).

## Preparing our rays

We have discussed the most theoretic part of training, no jokes. Now, it's time to implement it.

### Camera preparing

Initing ranges for view, and making position + direction for our camera

```
// clean back buffer with color
this->CleanBackbuffer( { 0.0f, 0.2f, 0.0f } );

const Math::int2& sizeWindow = this->_Window.getSize();

// camera is looking along to 'Z' axis ( LEFT HAND system! ),
// BE double awere with this, all samples in internet use RIGHT-hand coordinate system!
float3 fCameraDirection = float3( 0.0f, 0.0f, 1.0f );

// current camera position( a little bit back, to see sphere fully )
const float3& camera_pos = { 0.0f,0.0f,-1.0f };

const float aspect_ratio = float( sizeWindow.x ) / float( sizeWindow.y );
float fMiddleAngle = 90.0f;
float fov = 45.0f;
float fHalffov = fov * 0.5f;

float cosmin = cosf( Math::AngleToRadian( fMiddleAngle - fHalffov ) );
float cosmax = cosf( Math::AngleToRadian( fMiddleAngle + fHalffov ) );

float3 hor_start( cosmax * aspect_ratio, 0.0f, 0.0f );
float3 hor_End( cosmin * aspect_ratio, 0.0f, 0.0f );

float3 ver_start( 0.0f, cosmin, 0.0f );
float3 ver_End( 0.0f, cosmax, 0.0f );
```

## Iterating over pixel

I suggest you to iterate row by row, debugging of such iterating more obvious than column by column, but final choice by you. Sample below

```
// moving from LEFT TOP, to RIGHT BOTTOM
// row by row( not column by column! )
for( int j = 0; j < sizeWindow.y; ++j ) // Y axis
{
    for( int i = 0; i < sizeWindow.x; ++i ) // X axis
    {
        // calculating diff for our ray from top_left point of viewport

        float u = float( i ) / ( sizeWindow.x - 1 );
        float v = float( j ) / ( sizeWindow.y - 1 );
```

We know range in viewport, we know dt for vertical and horizontal movement, let's use them! Sample below

```
            // total distribution
            // x->   [ -1.0f , 1.0f ]
            // y->   [ 1.0f , -1.0f ]
            float3 dir = Math::Lerp( hor_start, hot_End, u ) + Math::Lerp( ver_start, ver_End, v ) + fCameraD

            Ray r;
            r.setPosition( camera_pos );
            r.setDirection( dir );

            // calculating color from ray
            float3 col = this->RayTracing( r );

            //// clamping extra energy!
            col.x = Math::Saturate( col.x );
            col.y = Math::Saturate( col.y );
            col.z = Math::Saturate( col.z );


            // put color in to backbuffer
            unsigned int uiIndex = i + j * sizeWindow.x;
            this->_Backbuffer[ uiIndex ] = col;
        }
    }
}
```

## Tasks

### Making horizontal gradient

You need to add a functional in Application::RayTracing to draw gradient color. Left side of screen → White color, Right size → fully Black. Range value for lerp between two color - .x of normalized ray direction.

NOTE: do not forget to convert [ -1, 1 ] range in to [ 0, 1 ] one.

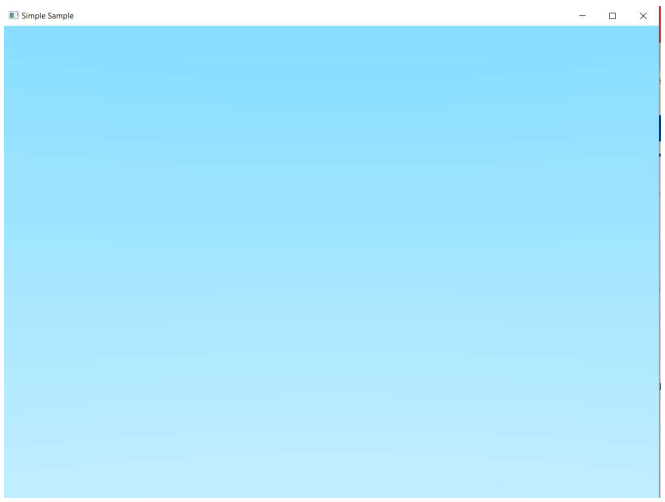
### Making vertical gradient

Add method in Application to draw vertical gradient with next colors. Top → White, Bottom → Black.

Note: be double aware of vertical NDC range.

Alter color for vertical gradient into light blue → white. You can notice, that changing red channel you change blue one. This is not a bug, Win32 uses BGR scheme instead of RGB. Fix should be trivial in code to use obtain expected behavior with float3 as color representation. One more time, you need to fix RGB→BGR!

Final result can be like.




No labels