Contents

Chapter I

Introduction

When it comes to rendering 3-dimensional computer-generated images there are 2 possible approaches: "`Rasterization`", which is used by almost all graphic engines because of its efficiency and "`Ray Tracing`."

The "`Ray Tracing`" method, developed for the first time in 1968 (but improved upon since) is even today more expensive in computation than the "`Rasterization`" method. As a result, it is not yet fully adapted to real time use-cases but it produce a much higher degree of visual realism.

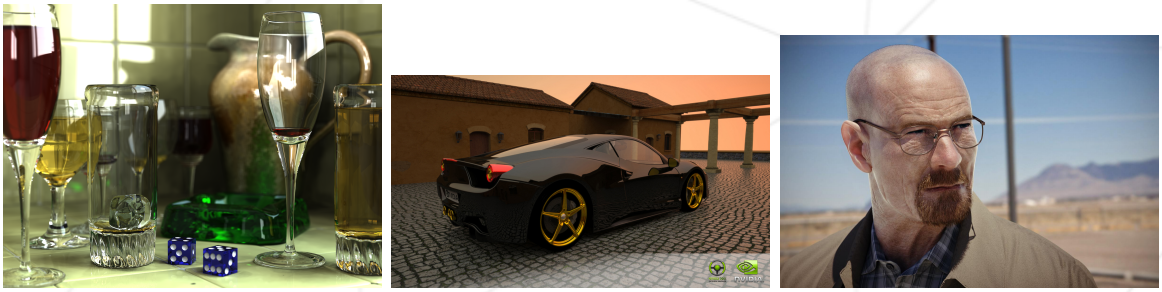


Figure I.1: The pictures above are rendered with the ray tracing technique. Impressive isn't it?

Before you can even begin to produce such high-quality graphics, you must master the basics: the `miniRT` is your first ray tracer coded in `C`, normed and humble but `functionnal`.

The main goal of `miniRT` is to prove to yourself that you can implement any mathematics or physics formulas without being a mathematician, we will only implement the most basics ray tracing features here so just keep calm, take a deep breath and `don't panic!` After this project you'll be able to show off nice-looking pictures to justify the number of hours you're spending at school...

Chapter II

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a `0` if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}` if the subject does not specify anything else. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

Mandatory part - miniRT

Program name	`miniRT`
Turn in files	`All your files`
Makefile	`all, clean, fclean, re, bonus`
Arguments	`a scene in format *.rt`
External functs.	• `open, close, read, write, printf, malloc, free, perror, strerror, exit` • `All functions of the math library (-lm man man 3 math)` • `All functions of the MinilibX`
Libft authorized	`Yes`
Description	`The goal of your program is to generate images using the Raytracing protocol. Those computer-generated images will each represent a scene, as seen from a specific angle and position, defined by simple geometric objects, and each with its own lighting system.`

The constraints are as follows:

- You `must` use the `miniLibX`. Either the version that is available on the operating system, or from its sources. If you choose to work with the sources, you will need to apply the same rules for your `libft` as those written above in `Common Instructions` part.
- The management of your window must remain fluid: switching to another window, minimization, etc..
- When you change the resolution of the window, the content of the window must remain unchanged and be adjusted accordingly.
- You need at least these 3 simple geometric objects: plane, sphere, cylinder.

- If applicable, all possible intersections and the inside of the object must be handled correctly.
- Your program must be able to resize the object's unique properties: diameter for a sphere and the width and height for a cylinder.
- Your program must be able to apply translation and rotation transformation to objects, lights and cameras (except for spheres and lights that cannot be rotated).
- Light management: spot brightness, hard shadows, ambiance lighting (objects are never completely in the dark). You must implement Ambient and diffuse lighting.
- the program displays the image in a window and respect the following rules:
  - Pressing `ESC` must close the window and quit the program cleanly.
  - Clicking on the red cross on the window's frame must close the window and quit the program cleanly.
  - The use of `images` of the `minilibX` is strongly recommended.
- Your program must take as a first argument a scene description file with the `.rt` extension.
  - Each type of element can be separated by one or more line break(s).
  - Each type of information from an element can be separated by one or more space(s).
  - Each type of element can be set in any order in the file.
  - Elements which are defined by a capital letter can only be declared once in the scene.

- Each element first's information is the type identifier (composed by one or two character(s)), followed by all specific information for each object in a strict order such as:
- **Ambient lightning:**

```
A 0.2 255,255,255
```

  - identifier: **A**
  - ambient lighting ratio in range [0.0,1.0]: **0.2**
  - R,G,B colors in range [0-255]: **255, 255, 255**
- **Camera:**

```
C -50.0,0,20 0,0,1 70
```

  - identifier: **C**
  - x,y,z coordinates of the view point: **-50.0,0,20**
  - 3d normalized orientation vector. In range [-1,1] for each x,y,z axis: **0.0,0.0,1.0**
  - FOV : Horizontal field of view in degrees in range [0,180]: **70**
- **Light:**

```
L -40.0,50.0,0.0 0.6 10,0,255
```

  - identifier: **L**
  - x,y,z coordinates of the light point: **-40.0,50.0,0.0**
  - the light brightness ratio in range [0.0,1.0]: **0.6**
  - (unused in mandatory part)R,G,B colors in range [0-255]: **10, 0, 255**
- **Sphere**:

```
sp 0.0,0.0,20.6 12.6 10,0,255
```

  - identifier: **sp**
  - x,y,z coordinates of the sphere center: **0.0,0.0,20.6**
  - the sphere diameter: **12.6**
  - R,G,B colors in range [0-255]: **10, 0, 255**

- **Plane:**

```
pl  0.0,0.0,-10.0  0.0,1.0,0.0  0,0,225
```

  - identifier: **pl**
  - x,y,z coordinates of a point in the plane: **0.0,0.0,-10.0**
  - 3d normalized normal vector. In range [-1,1] for each x,y,z axis: **0.0,1.0,0.0**
  - R,G,B colors in range [0-255]: **0,0,225**

- **Cylinder:**

```
cy  50.0,0.0,20.6  0.0,0.0,1.0  14.2  21.42  10,0,255
```

  - identifier: **cy**
  - x,y,z coordinates of the center of the cylinder: **50.0,0.0,20.6**
  - 3d normalized vector of axis of cylinder. In range [-1,1] for each x,y,z axis: **0.0,0.0,1.0**
  - the cylinder diameter: **14.2**
  - the cylinder height: **21.42**
  - R,G,B colors in range [0,255]: **10, 0, 255**

- Example of the mandatory part with a minimalist **.rt** scene:

```
A   0.2                                 255,255,255

C   -50,0,20      0,0,0       70
L   -40,0,30                  0.7           255,255,255

pl  0,0,0         0,1.0,0                   255,0,225
sp  0,0,20                    20            255,0,0
cy  50.0,0.0,20.6 0,0,1.0     14.2  21.42   10,0,255
```

- If any misconfiguration of any kind is encountered in the file the program must exit properly and return "Error\n" followed by an explicit error message of your choice.
- For the defense, it would be ideal for you to have a whole set of scenes with the focus on what is functional, to facilitate the control of the elements to create.

Chapter IV

Bonus part

The `Ray-Tracing` technique could handle many more things like reflection, transparency, refraction, more complex objects, soft shadows, caustics, global illumination, bump mapping, .obj file rendering etc..

But for the `miniRT` project, we want to keep things simple for your first raytracer and your first steps in CGI.

So here is a list of few simple bonuses you could implement, if you want to do bigger bonuses we strongly advise you to recode a new ray-tracer later in your developer life after this little one is finished and fully functionnal.

Figure IV.1: A spot, a space skybox and a shiny earth-textured sphere with bump-maping

`Bonuses will be evaluated only if your mandatory part is PERFECT.`
`By PERFECT we naturally mean that it needs to be complete, that it`
`cannot fail, even in cases of nasty mistakes like wrong uses etc.`
`It means that if your mandatory part does not obtain ALL the points`
`during the grading, your bonuses will be entirely IGNORED.`

Bonus list:

- Add specular reflection to have a full Phong reflection model.
- Color disruption: checkerboard.
- Colored and multi-spot lights.
- One other 2nd degree object: Cone, Hyperboloid, Paraboloid..
- Handle bump map textures.

You are allowed to use other functions and add features to your scene description to complete the bonus part as long as their use is justified during your evaluation. You are also allowed to modify the expected scene file format to fit your needs. Be smart!

Chapter V

Examples

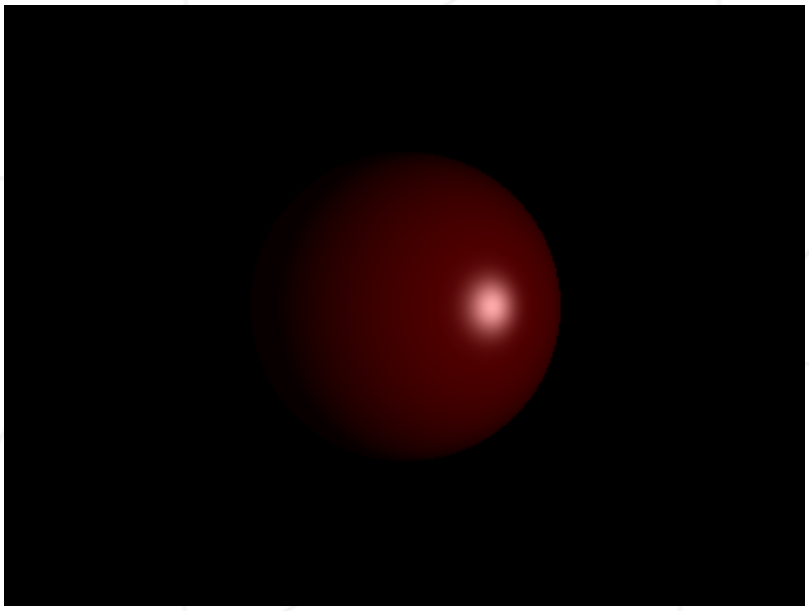


Figure V.1: A sphere, one spot, some shine (optional)

Figure V.2: A cylinder, one spot

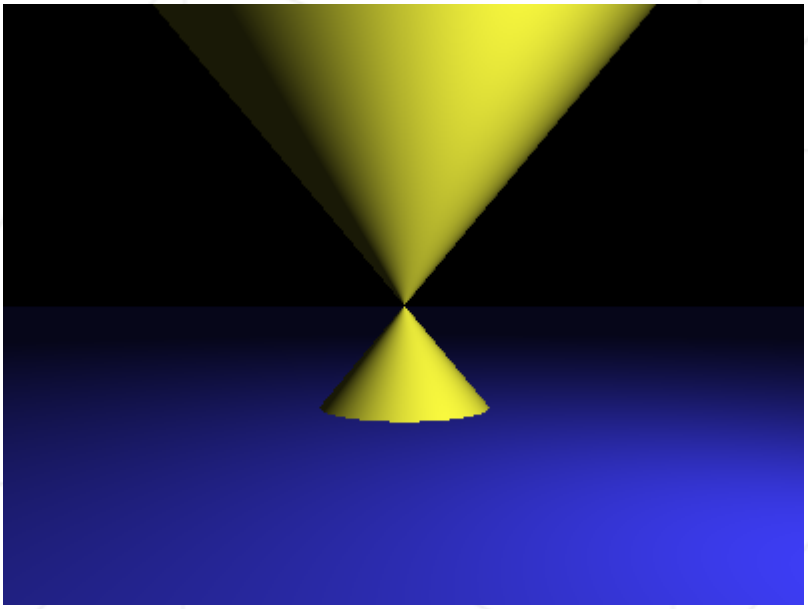


Figure V.3: A cone (optional), a plane, one spot

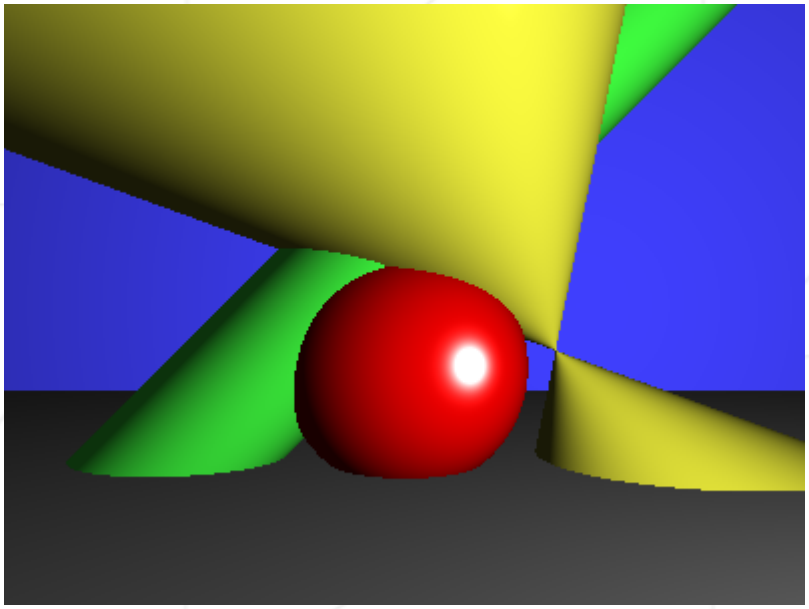


Figure V.4: A bit of everything, including 2 planes

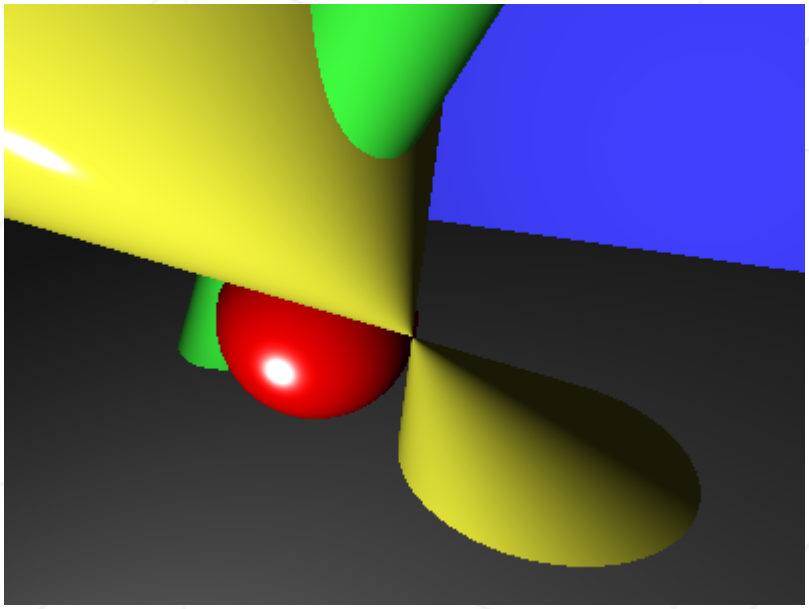


Figure V.5: Same scene different camera

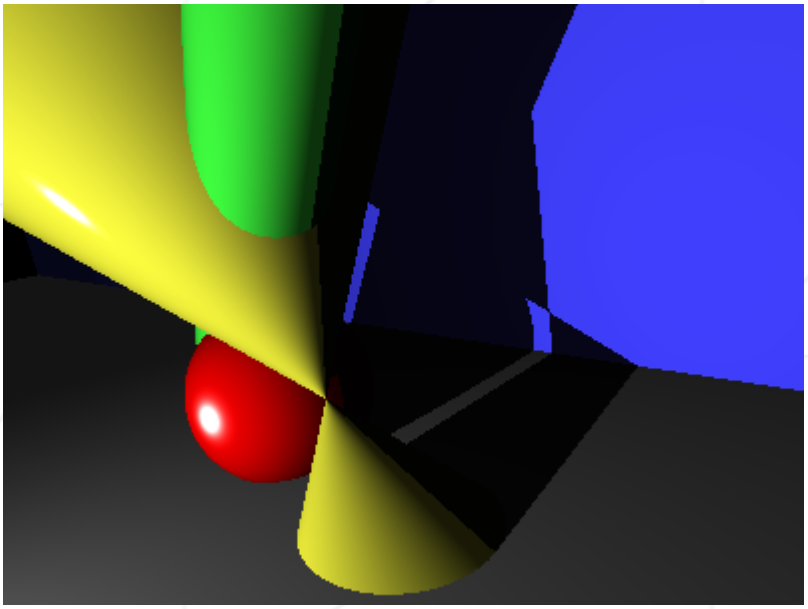


Figure V.6: This time with shadows

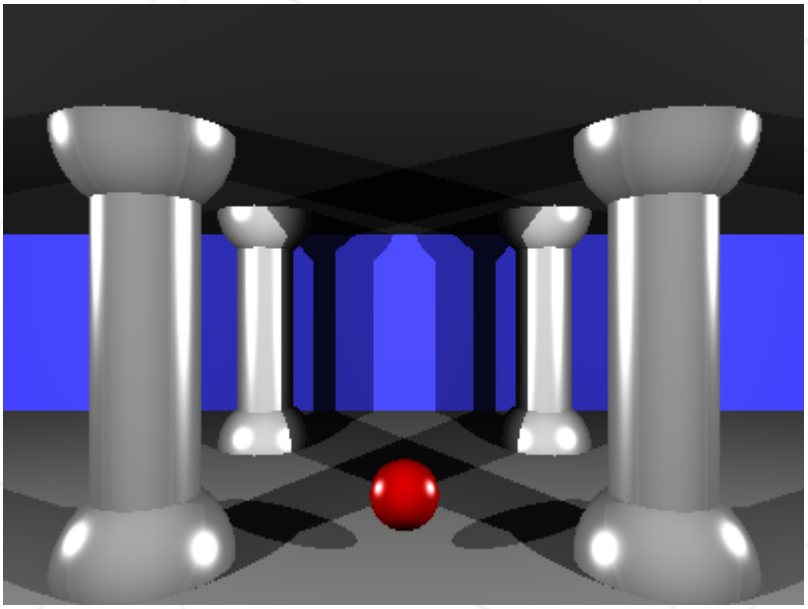


Figure V.7: With multiple spots

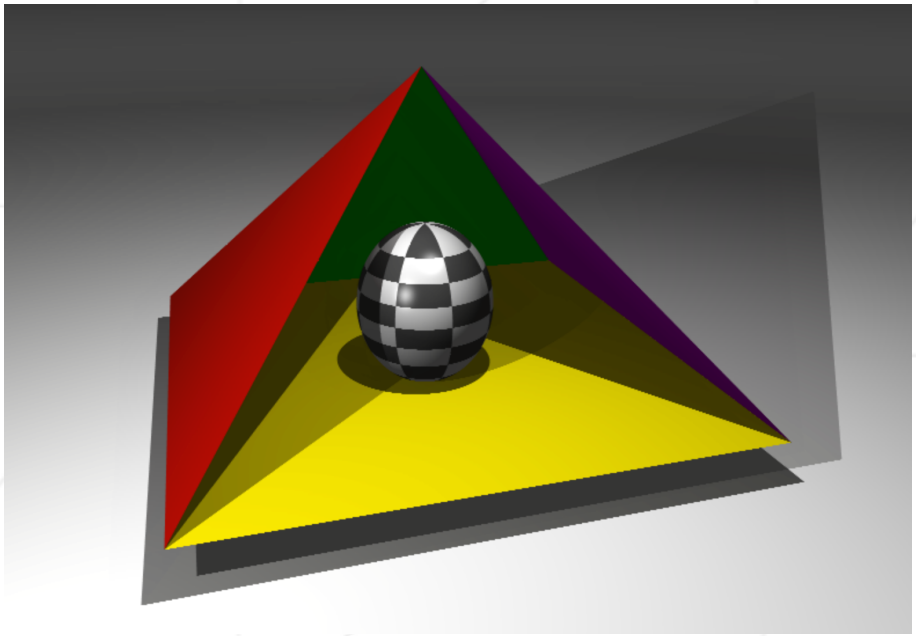


Figure V.8: And finally with multiple spots and a shiny checkered (optional) sphere in the middle

Chapter VI

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

```
???????????  XXXXXXXXXX = $3$$f2c51fc265cf7e70865cb1bd24214beb
```

Chapter3. レイと物体の交差判定

knzw.tech/raytracing

交差判定について

レイトレーシング法を用いるにはレイ（半直線）と物体の交点を求める必要がある．

現実の物体の形状はきわめて複雑であり，レイとの交点を求めることはおろか， その形状をプログラム中で利用できるように記述することも容易ではない．

そこで，ここでは交点を求める物体として**球**と**平面**(無限平面)を扱う． これらの物体とレイの交点を求めるのは比較的容易である．

さまざまな物体の方程式

ベクトル方程式について

曲面や点の位置を示すのにベクトル方程式を用いると便利な場合がある．

ベクトル方程式とは，等式の中に**ベクトルが出現する方程式**である．
もっとも簡単なベクトル方程式は以下のようなものである．

$\vec{a} = \vec{b}$

この方程式はベクトル$\vec{a}$と$\vec{b}$が，同じ向きと大きさをもつことを意味する．

ベクトル方程式で注意するべきことは，**両辺ともスカラーであるか，両辺ともベクトルである方程式しか意味を持たない**ということである（スカラーとはベクトルに対して単一の値のみを持つ量を意味する．例えば，ベクトルの成分はスカラーである）．
たとえば，以下のような方程式の左辺がベクトル量で左辺がスカラー量であるような方程式は成立し得ない．

/(\|\|538\)/\|\|=42…?

ベクトルに対する演算には，結果がベクトルになる演算（実数倍，加算，減算，外積）と，結果がスカラーになる演算（ベクトルのノルム，内積など）がある．
以降でベクトル方程式をいろいろと変形する部分があるため，この点に注意すること．

本節の残りの部分では，ベクトル方程式を用いてレイ（半直線），球，平面を表現する．
ただし，わかりやすさのため初等数学で用いる解析表示も併記する．

レイ（半直線）の方程式

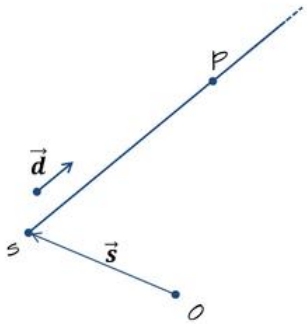


図1. 半直線

レイ（半直線）は直線と似ているが，始点を持ち始点から無限に伸びる直線である(図1)． レイは**始点sと方向ベクトル$\vec{d}$**で定義される．

レイの方程式は媒介変数$t(t \geq 0)$を用いて以下のように表すことができる．

⎧⎩⎨x=sx+tdxy=sy+tdyz=sz+tdz

(添え字つきのs,dはそれぞれ始点の位置ベクトルと方向ベクトルの成分である．)
(直線の方程式は，媒介変数を消去して$x-sxdx=y-sydy=z-szdz$と表す場合もある．)

ベクトル方程式で表すと以下のようになる．

$\vec{p}=\vec{s}+t\vec{d}$

($\vec{s}$は点sの位置ベクトルである．)

文章で書けば，*半直線上の点pの位置ベクトルは始点の位置ベクトルに方向ベクトルの実数倍を足したものとして表現できる*，ということである(図2)．

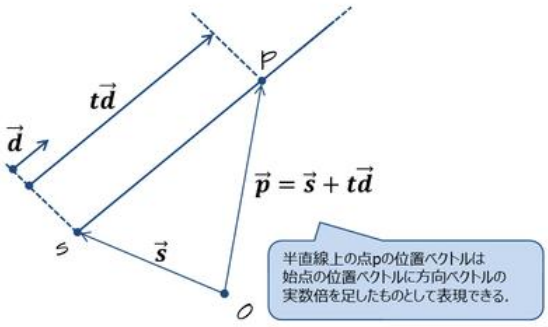


図2. 半直線上の点

平面の方程式

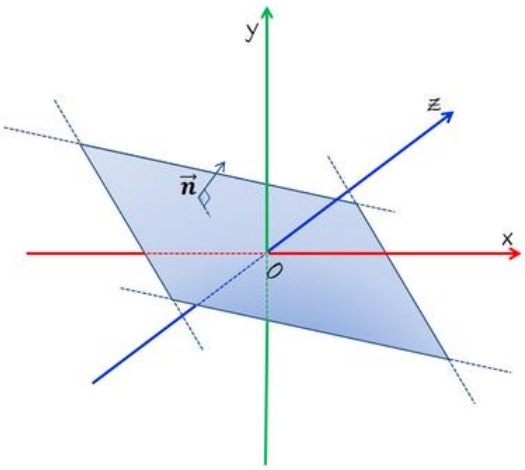


図3. 原点を通る平面

原点を通る，平面の方程式は以下である．

$ax+by+cy=0$

これはa,b,cを法線ベクトル(平面に直交するベクトル)の成分に持つ平面である．

この式は以下のように主張している．

この(曲)面に属する点$\vec{p}$=(x,y,z)と法線ベクトル$\vec{n}$=(a,b,c)の内積は0である(図4)．

このことをベクトル方程式で表現すると以下のようになる．

$(\vec{p} \cdot \vec{n})=0$

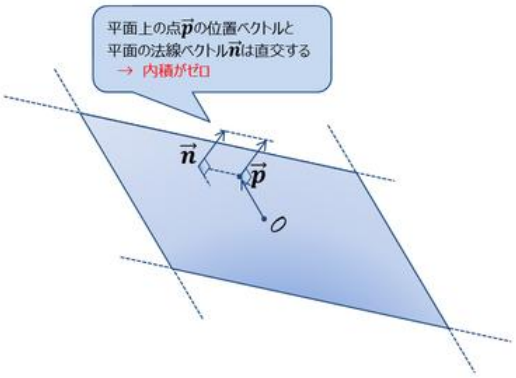


図4. 平面上の点

球の方程式

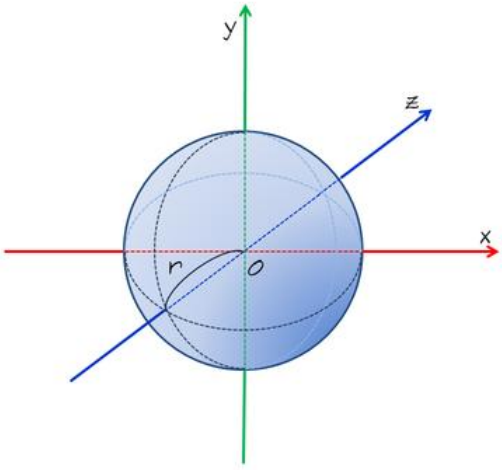


図5. 原点を中心とする球

原点を中心とする，球の方程式は以下である．

$x2+y2+z2=r2$

この式は以下のように主張している．

この曲面上の点p=(x,y,z)と原点の距離はrである．

言い換えると

この曲面上の点pの位置ベクトル$\vec{p}$=(x,y,z)のノルム(長さ)はrである(図6)．

このことをベクトル方程式で表現すると以下のようになる．

$|\vec{p}|=r$

のちの計算でこの式を用いる場合，ベクトルのノルムには平方根が入るため両辺を二乗する場合がある．

$|\vec{p}|^2=r^2$

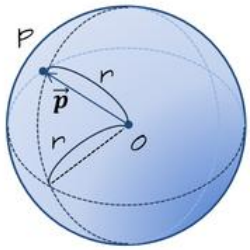


図6. 球面上の点

レイと物体の交差判定

実際にレイと物体の交差判定を行うには**上述した方程式を連立しレイの方程式中のtの値を調べればよい**．

レイと平面の交差判定

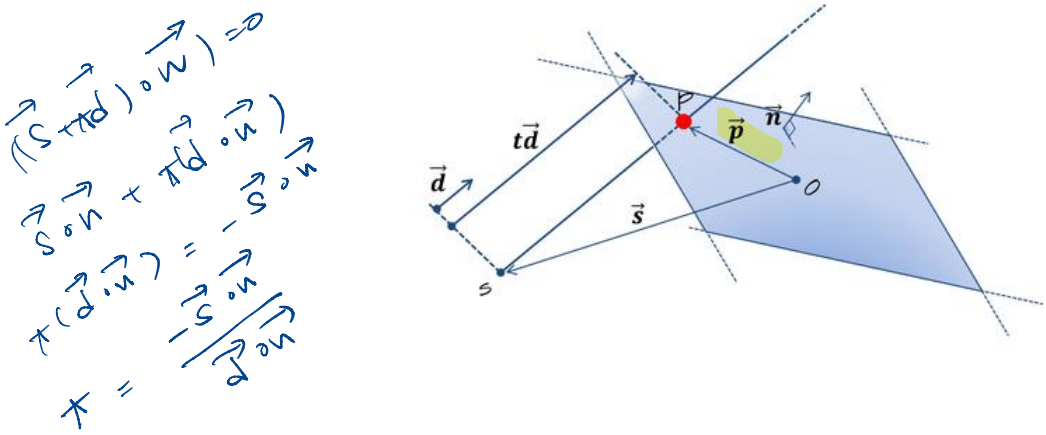


図7. レイと平面の交差判定

- レイのベクトル方程式：$\vec{p}=\vec{s}+t\vec{d}$ …(1)
- 平面のベクトル方程式：$(\vec{p}\cdot\vec{n})=0$…(2)

式(2)に式(1)を代入する(内積は分配法則が成り立つことに注意せよ)．

$((\vec{s}+t\vec{d})\cdot\vec{n})=0$…(3)
式(3)をtについて解く．

$((\vec{s}+t\vec{d})\cdot\vec{n})=0(\vec{s}\cdot\vec{n})+t(\vec{d}\cdot\vec{n})=0t(\vec{d}\cdot\vec{n})=-(\vec{s}\cdot\vec{n})t=-(\vec{s}\cdot\vec{n})(\vec{d}\cdot\vec{n})$

tの値の解釈

tの値は以下のように解釈できる．

まず，tの分母がゼロの場合，**すなわち$(\vec{d}\cdot\vec{n})=0$のときレイと平面は交点を持たない．**

二つのベクトルの内積がゼロということは，**それらのベクトルが直交している**ということである（レイの方向ベクトル$\vec{d}$と 平面の法線ベクトル$\vec{n}$が直交しているならば交点は持たない）．

それ以外の場合は以下である．

- $t>0$のとき：レイは平面と交点を持つ．交点は$\vec{s}+t\vec{d}$である．
- $t=0$のとき：レイの始点は平面上にある．
- $t<0$のとき：平面はレイの法線ベクトルの逆方向，すなわちレイの始点よりも後ろにある．したがって交点を持たない．

レイと球の交差判定

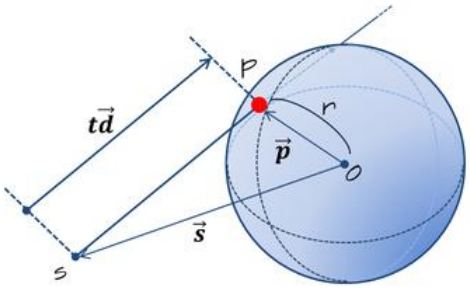


図8. レイと球の交差判定

- レイのベクトル方程式：$\vec{p}=\vec{s}+t\vec{d}$ …(1)
- 球のベクトル方程式：$|\vec{p}|2=r2$…(2)

式(2)に式(1)を代入する．

$|\vec{s}+t\vec{d}|2=r2$…(3)
式(3)をtについて解く（同じベクトル自身との内積は，そのベクトルのノルム（長さ）の二乗となることに注意せよ）．

$|\vec{s}+t\vec{d}|2=r2(\vec{s}+t\vec{d})\cdot(\vec{s}+t\vec{d})=r2(\vec{s}\cdot\vec{s})+2(\vec{s}\cdot\vec{d})t+(\vec{d}\cdot\vec{d})t2=r2|\vec{s}|2+2(\vec{s}\cdot\vec{d})t+|\vec{d}|2t2=r2|\vec{d}|2t2+2(\vec{s}\cdot\vec{d})t+|\vec{s}|2=r2|\vec{d}|2t2+2(\vec{s}\cdot\vec{d})t+|\vec{s}|2-r2=0$…(4)
式(4)はtに関する二次方程式となっている．すなわち，A,B,Cを以下のようにおくと，

$ABC===|\vec{d}|22(\vec{s}\cdot\vec{d})|\vec{s}|2-r2$

式(4)は以下の形となっている．

$At2+Bt+C=0$…(4)′

したがってtの根は以下である．

$t=-B\pm B2-4AC\sqrt{2A}$

※ この公式に上述のA,B,Cの値を当てはめてさらに展開する**必要はない**.
なぜなら、これらの式を計算機上で計算する場合はA,B,Cを変数として用意してtの計算を行えばよいためである.

tの値の解釈

tの値は以下のように解釈できる．

まず，判別式$D=B2-4AC$のときtは実根を持たない．**この場合は，レイと球が交差しないことを意味する．**

D=0のとき，レイは球とただ一つの交点を持つ．
D>0のとき，レイは球と二つの交点を持つ．

- t>0のとき：レイは球と交点を持つ．交点は$\vec{s}+t\vec{d}$ である．
- t=0のとき：レイの始点は球面上にある．
- t<0のとき：球はレイの法線ベクトルの逆方向，すなわちレイの始点よりも後ろにある．したがって交点を持たない．

課題

課題1-4 任意の点を通る平面とレイの交点

(作業時間目安：15分)

平面の方程式は以下であると前節で述べた．

$ax+by+cy=0 \Leftrightarrow (\vec{\mathbf{p}}\cdot\vec{\mathbf{n}})=0$

これは平面が原点を通る場合の方程式である．一般に任意の点$\mathbf{pc}\rightarrow=(cx,cy,cz)$を通る平面の方程式は 以下のように表される．

$a(x-cx)+b(y-cy)+c(z-cz)=0$
ベクトル方程式として表現すると，

$(\vec{\mathbf{p}}-\mathbf{pc}\rightarrow)\cdot\vec{\mathbf{n}}=0 \ldots(1)$
となる．

式(1)とレイの方程式$\vec{\mathbf{p}}=\vec{\mathbf{s}}+t\vec{\mathbf{d}}$ …(2)を連立させ，tの一般解を求めよ．

課題1-5 任意の点を中心とする球とレイの交点

(作業時間目安：15分)

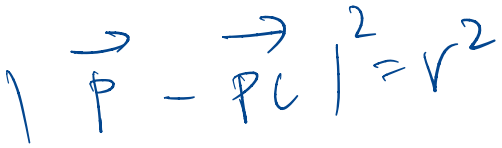
球の方程式は以下であると前節で述べた．

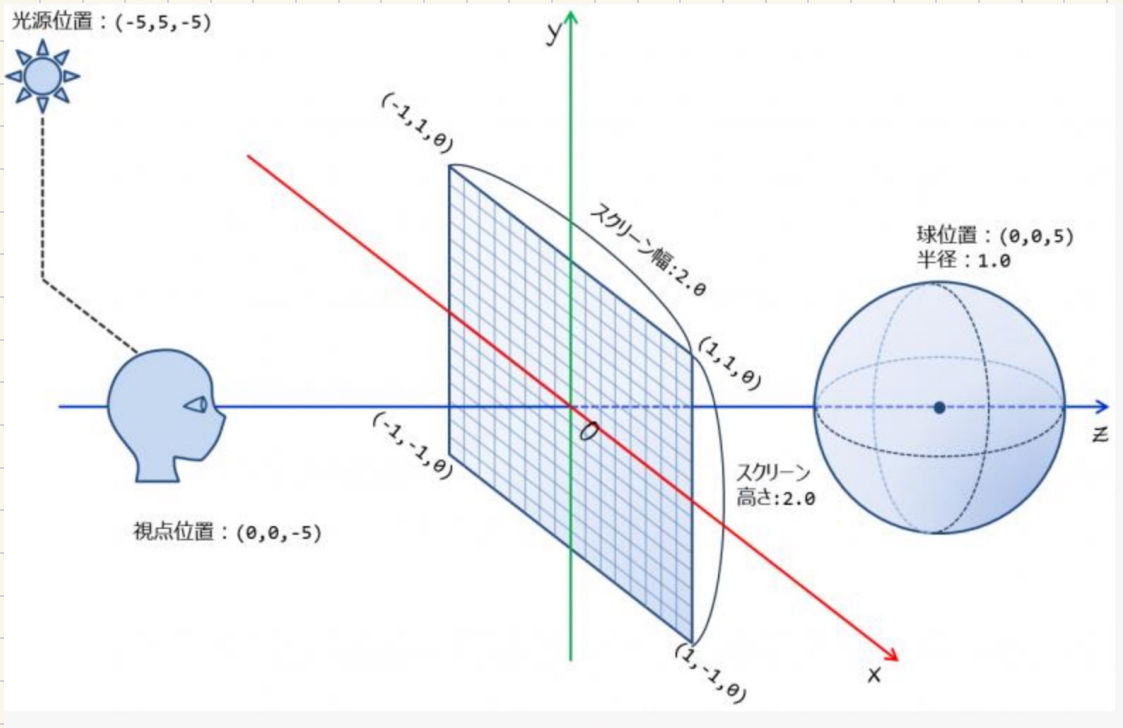
$x2+y2+z2=r2 \Leftrightarrow |\vec{\mathbf{p}}|2=r2$
これは球が原点を中心とする場合の方程式である．一般に任意の点$\mathbf{pc}\rightarrow=(cx,cy,cz)$を中心とする球の方程式は 以下のように表される．

$(x-cx)2+(y-cy)2+(z-cz)2=r2$
ベクトル方程式として表現すると，

$|\vec{\mathbf{p}}-\mathbf{pc}\rightarrow|2=r2 \ldots(1)$
となる．

式(1)とレイの方程式$\vec{\mathbf{p}}=\vec{\mathbf{s}}+t\vec{\mathbf{d}}$ …(2)を連立させ，tの一般解を求めよ．




光源位置：(-5,5,-5)
y
(-1,1,0)
スクリーン幅:2.0
(1,1,0)
球位置：(0,0,5)
半径：1.0
O
z
(-1,-1,0)
スクリーン
高さ:2.0
視点位置：(0,0,-5)
(1,-1,0)
x

[ベクトルの内積と外積]

dot　$\vec{a}\cdot\vec{b}=|\vec{a}||\vec{b}|\cos\theta$

2つのベクトルの位置関係

$\vec{a}\cdot\vec{b}>0$　$\vec{b}$は$\vec{a}$と同じ方を向く

$\vec{a}\cdot\vec{b}<0$　逆向き

$\vec{a}\cdot\vec{b}=0$　垂直 直行

$\vec{a}\cdot\vec{b}=|\vec{a}||\vec{b}|$　ベクトルが平行

$|\vec{a}|\,|\vec{b}|$を正規化すると？

$\vec{a}\cdot\vec{b}=|\vec{a}||\vec{b}|\cos\theta$

$\hat{a}\cdot\hat{b}=1\times1\times\cos\theta$

$\hat{a}\cdot\hat{b}=\cos\theta$

外積(クロス)　$\vec{a}\times\vec{b}=|\vec{a}||\vec{b}|\sin(\theta)$

ベクトルの左右関係

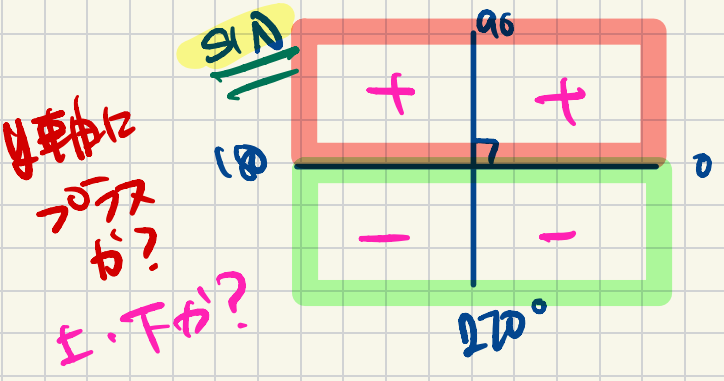
あるベクトルが左にくる？右にくる？

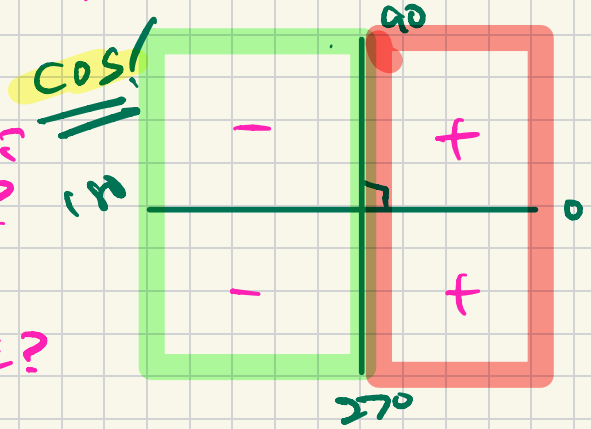
$\vec{a}\times\vec{b}>0$　$\vec{b}$は$\vec{a}$の左側

$\vec{a}\times\vec{b}<0$　$\vec{b}$は$\vec{a}$の右側

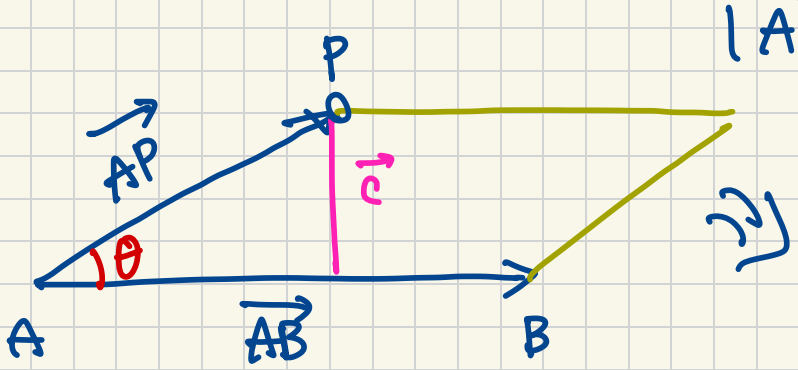
$\vec{a}\times\vec{b}=0$　$\vec{a}$と$\vec{b}$は平行（0°のsinは0）

	sin	
0°	0	プラス
90°	1	
180°	0	
270°	−1	マイナス
0	0	

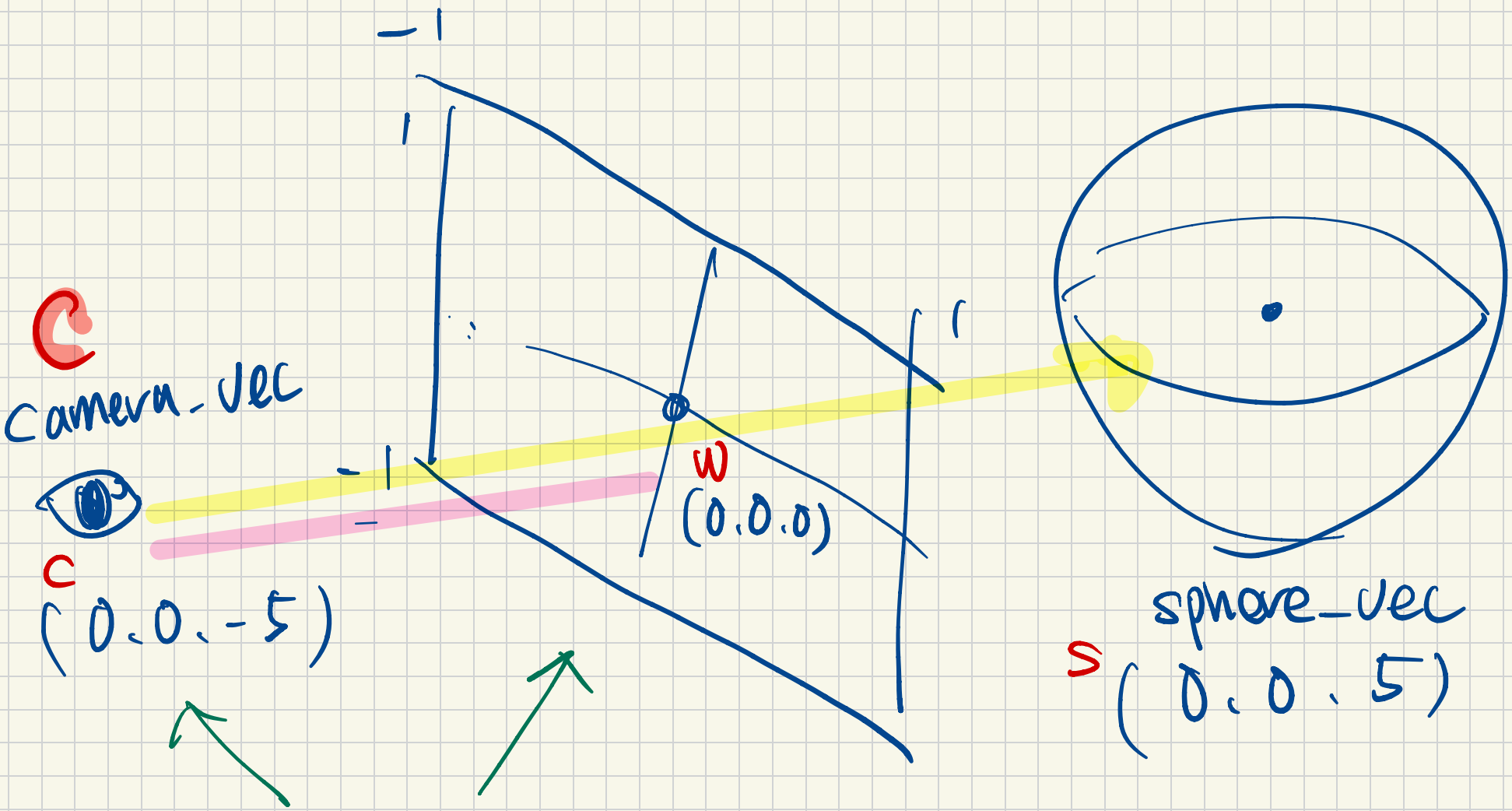




[外積]



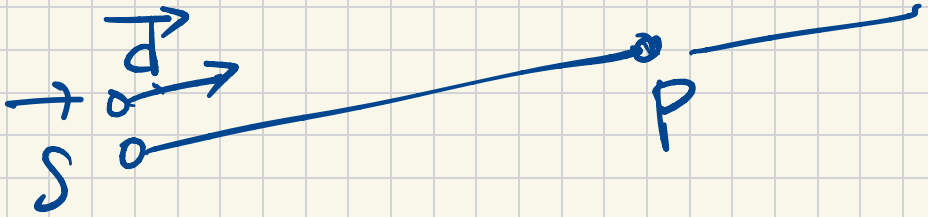
$|\vec{AP}|\,|\vec{AB}|\sin\theta$



$$\overrightarrow{CW} = \overrightarrow{OW} - \overrightarrow{OC} = \vec{W} - \vec{C} = (0,0,5)$$

〈それぞれの物体のベクトル方程式〉

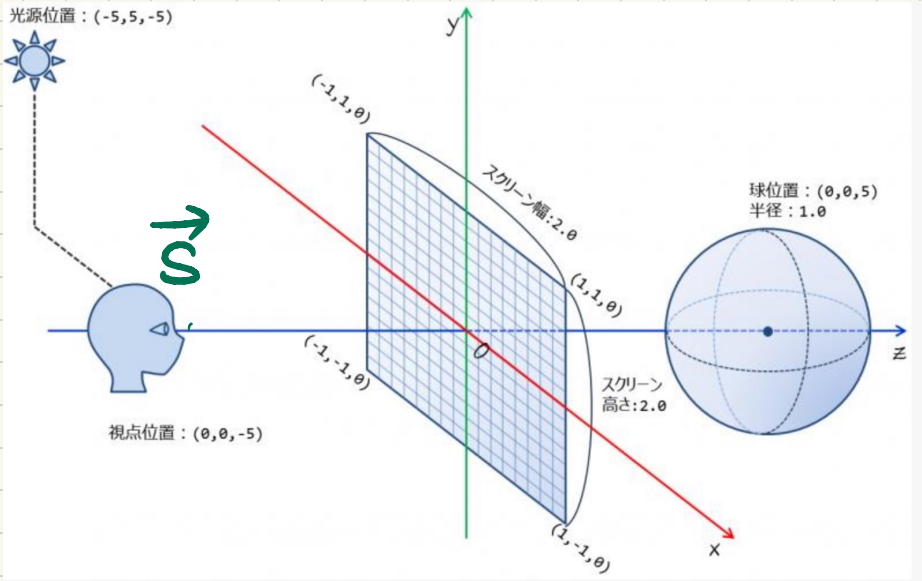
〈Ray（半直線）〉



$$\vec{P} = \vec{s} + t\vec{d}$$

始点の位置ベクトル

方向ベクトルの実数倍



<平面のベクトル方程式> $ax + by + cz + d = 0$

$\therefore$ a, b, c = 法線ベクトル = $\vec{n}$ (a, b, c)

d = 原点からの距離

x, y, z = 面に属する点 = $\vec{P}(x, y, z)$

$\Rightarrow (\vec{P} \circ \vec{n}) = 0$

[レイと平面の交差判定]

Ray $\vec{P} = \vec{s} + t\vec{d}$

平面 $(\vec{P} \circ \vec{n}) = 0$

$(\vec{s} + t\vec{d}) \circ \vec{n} = 0$

$\vec{s} \circ \vec{n} + t(\vec{d} \circ \vec{n}) = 0$

$t(\vec{d} \circ \vec{n}) = -(\vec{s} \circ \vec{n})$

$t = -\dfrac{(\vec{s} \circ \vec{n})}{(\vec{d} \circ \vec{n})}$

① 分母 $\vec{d} \circ \vec{n} = 0$ は使えない

② $t > 0$ 交点 @ $\vec{s} + t\vec{d}$

$t = 0$ レイの始点 = 平面上

$t < 0$ レイの行き = 交点なし

平面ベクトル(原点外)との衝点.　　点 $\vec{P_c} = (C_x, C_y, C_z)$

$a(x-C_x)+b(y-C_y)+z(z-C_z)=0$

$\hookrightarrow (\vec{P}-\vec{P_c})\cdot\vec{n}=0$

$\vec{P}=\vec{s}+t\vec{d}$

$(\vec{s}+t\vec{d}-\vec{P_c})\cdot\vec{n}=0$

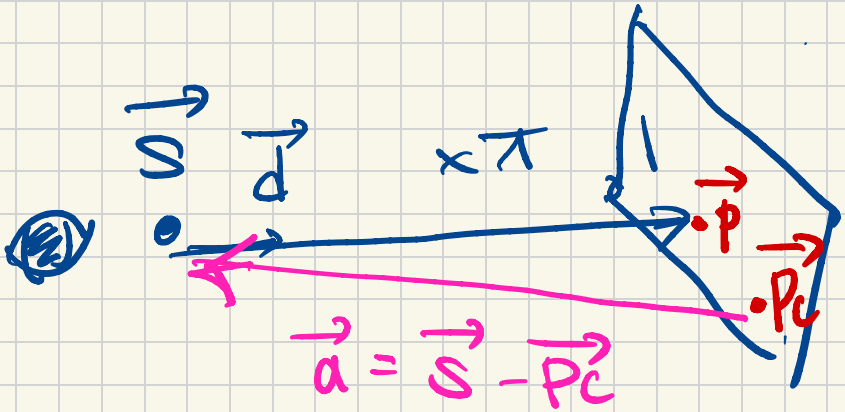
$\vec{s}-\vec{P_c}=\vec{a}$

$(\vec{a}+t\vec{d})\cdot\vec{n}=0$

$\vec{a}\cdot\vec{n}+t(\vec{d}\cdot\vec{n})=0$

$t(\vec{d}\cdot\vec{n})=-(\vec{a}\cdot\vec{n})$

$$t=\frac{-(\vec{a}\cdot\vec{n})}{(\vec{d}\cdot\vec{n})}$$



① $\vec{d}\cdot\vec{n}=0$ は解なし

② $t>0$　衝点 $\vec{s}+t\vec{d}$

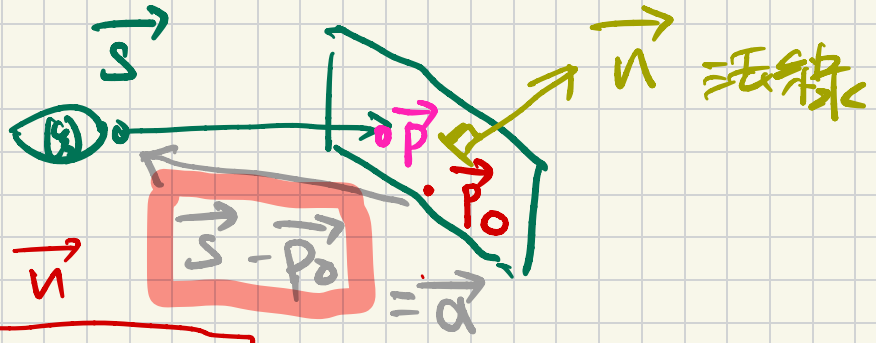
$t=0$　衝点 $\vec{s}$

$t<0$　なし

〈平面との交差（原点ではない）〉

Ray：$\vec{P} = \vec{S} + t\vec{d}$

平面：$\vec{P} \cdot \vec{n} = 0$ （原点）



$\vec{P} = (x, y, z)$

$\vec{P_0} = (x_0, y_0, z_0)$

$\vec{n} = (a, b, c)$

$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$

原点外 $(\vec{P} - \vec{P_0}) \cdot \vec{n} = 0$

$\vec{P} = \vec{S} + t\vec{d}$ だから

$(\vec{S} + t\vec{d} - \vec{P_0}) \cdot n = 0$

$(\vec{S} - \vec{P_0} + t\vec{d}) \cdot n = 0$

$\vec{a} = \vec{S} - \vec{P_0}$

$(\vec{a} + t\vec{d}) \cdot n = 0$

$(\vec{a} \cdot \vec{n}) + (\vec{d} \cdot \vec{n})t = 0$

$(\vec{d} \cdot \vec{n})t = -(\vec{a} \cdot \vec{n})$

$$t = \frac{-(\vec{a} \cdot \vec{n})}{(\vec{d} \cdot \vec{n})}$$

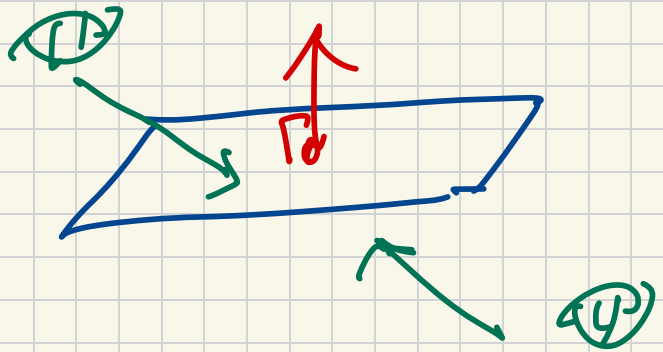
① 分母 $(\vec{d} \cdot \vec{n}) = 0$ は解なし

② $t = 0$ $\vec{P} = \vec{S}$

$t > 0$ $\vec{P} = \vec{S} + t\vec{d}$

$t < 0$ $\vec{P}$ は $\vec{S}$ より後ろ、交差なし

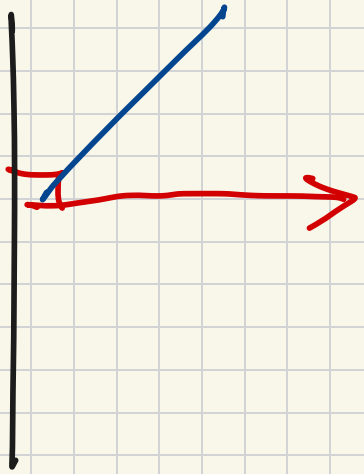
「平面の法線判定」



平面の法線と eyeadv の内積が

プラス = 同じ方向

マイナス = 逆方向



法線の cos = x

よって θ = 0 = 1 270 = 0

90 = 0

180 = -1

<球の方程式> $x^2+y^2+z^2=r^2$

点 $P=(x,y,z)$ と原点の距離は r

$|\vec{P}|=r$ $\quad |\vec{P}|^2=r^2$

<球と直線の交点>

直線 $\vec{P}=\vec{s}+t\vec{d}$

球 $|\vec{P}|^2=r^2$

$|\vec{s}+t\vec{d}|^2=r^2$ より

同じベクトルの内積は $|\ |^2$

$(\vec{s}+t\vec{d})\cdot(\vec{s}+t\vec{d})=r^2$

$(\vec{s}\cdot\vec{s})+2(\vec{s}\cdot\vec{d})t+(\vec{d}\cdot\vec{d})t^2=r^2$

$|\vec{s}|^2+2(\vec{s}\cdot\vec{d})t+|\vec{d}|^2t^2$

$|\vec{d}|^2t^2+2(\vec{s}\cdot\vec{d})t+|\vec{s}|^2-r^2=0$

t の二次方程式

$A=|\vec{d}|^2$

$B=2(\vec{s}\cdot\vec{d})$

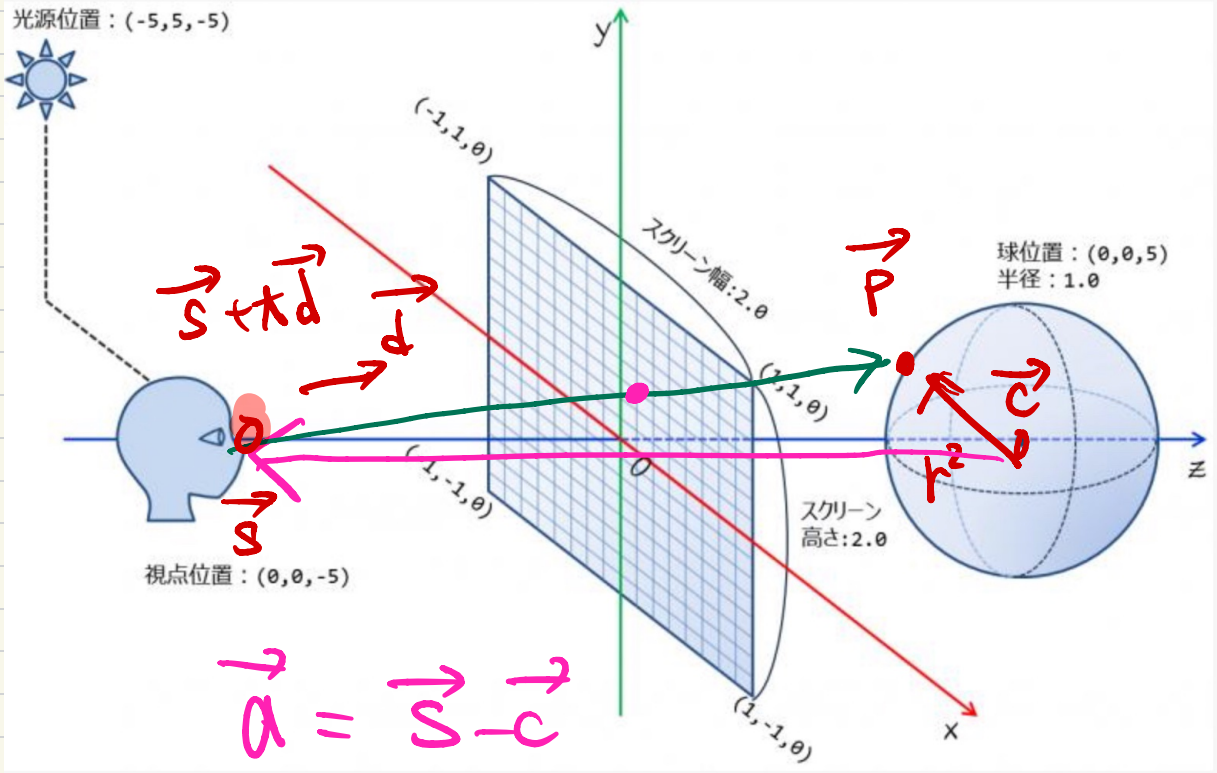
$C=|\vec{s}|^2-r^2$

$D=B^2-4AC$

$D>0$ は 2つの接点

$D=0$ は 1つの接点

$D<0$ は なし



$\vec{a} = \vec{s} - \vec{c}$

$\vec{P} = \vec{s} + t\vec{d}$ 线

$|\vec{P} - \vec{c}|^2 = r^2$ 球

$|\vec{s} + t\vec{d} - \vec{c}|^2$

$(\vec{s} - \vec{c} + t\vec{d}) \cdot (\vec{s} - \vec{c} + t\vec{d})$

$\vec{a}$

$(\vec{a} + t\vec{d}) \cdot (\vec{a} + t\vec{d}) = r^2$

$|\vec{a}|^2 + 2(\vec{a} \cdot \vec{d})t + |\vec{d}|^2 t^2 = r^2$

$|\vec{d}|^2 t^2 + 2(\vec{a} \cdot \vec{d})t + |\vec{a}|^2 - r^2 = 0$

A　B　C

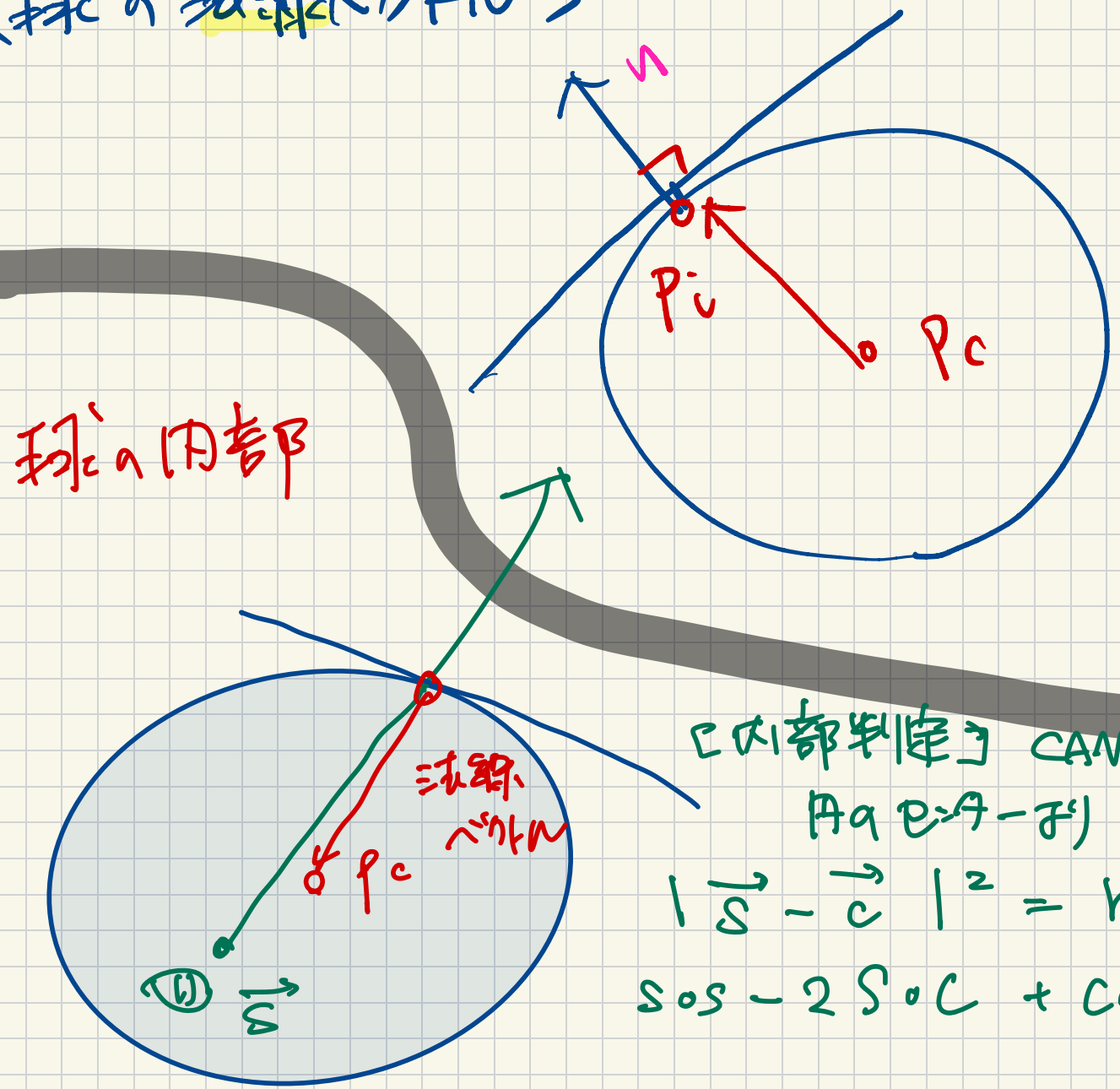
$\vec{a} = \vec{s} - \vec{c}$

$D = b^2 - 4ac$

$D > 0$ 时：

$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} =$ 大

<球の法線ベクトル>



$$\frac{\vec{P_i} - \vec{P_c}}{|\vec{P_i} - \vec{P_c}|}$$

球の法線

球の中心 $\vec{P_c}$ から交点 $\vec{P_i}$ へのベクトルを標準化

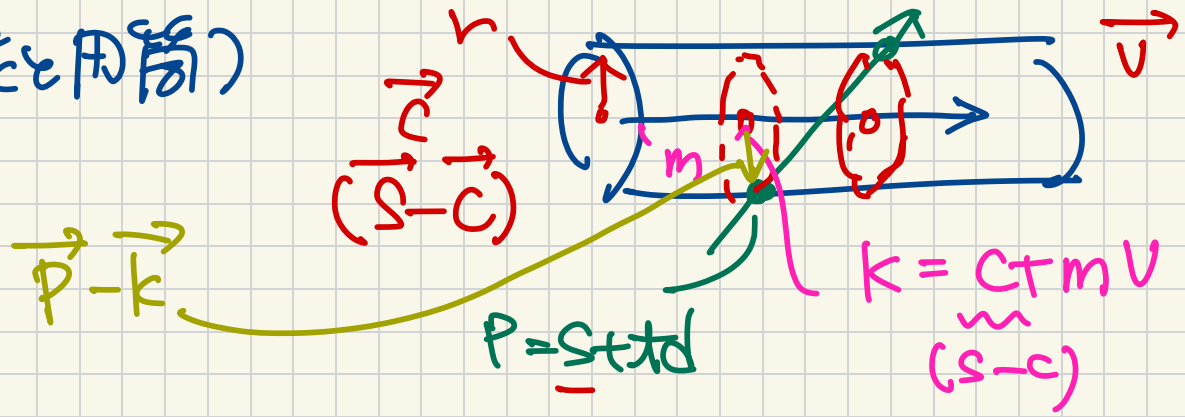
[内部判定] Cameraの pos が

円のセンターより r 以内

$$|\vec{S} - \vec{C}|^2 = r^2$$

$$S \cdot S - 2S \cdot C + C \cdot C - r^2 \leq 0$$

[光線と円筒]



$P=\vec{s}+t\vec{d}$

$k=\vec{s}-\vec{c}+m\vec{v}$

---

① $(P-k)\cdot(P-k)=r^2$

$P^2-2Pk+k^2$

$P^2=(S+td)^2=S^2+2t(S\cdot v)+t^2(d\cdot d)$

$=c^2+S^2+2t(S\cdot v)$

$2Pk=2\{(S+td)(S-c+mv)\}$

$2\{S^2-(S\cdot c)+m(S\cdot v)+t(d\cdot S)-t(d\cdot c)+tm(d\cdot v)\}$

② $(P-k)\cdot\vec{v}=0$

$\{(S+td)-(S-c+mv)\}\cdot v=0$

$(S+td-S+c-mv)\cdot v$

$t(d\cdot v)+(c\cdot v)-m(v\cdot v)=0$

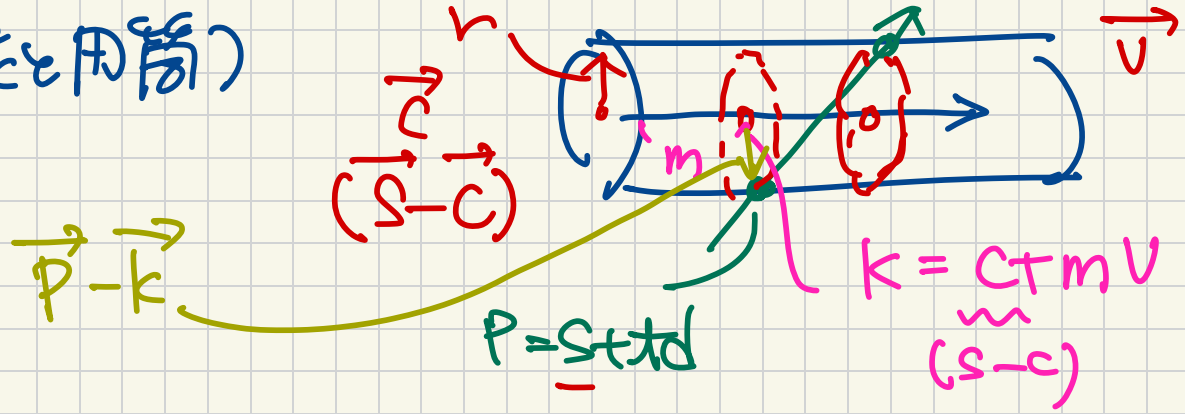
$m=t(d\cdot v)+(c\cdot v)$

$k^2=(S-c+mv)(S-c+mv)=S^2-(S\cdot c)+m(S\cdot v)-(c\cdot S)+(c\cdot c)-m(c\cdot v)$

$+m(v\cdot S)-m(v\cdot c)+m^2(v\cdot v)=S^2-2(S\cdot c)+2m(S\cdot v)+(c\cdot c)$

$-2m(c\cdot v)+m^2$

[光線と円筒]



$P=\vec{S}+t\vec{d}$

$k=\vec{S}-\vec{C}+m\vec{V}$

---

① $(P-k)\cdot(P-k)=r^2$

$P^2-2Pk+k^2$

$P^2=(S+td)^2=S^2+2t(S\cdot V)+t^2(d\cdot d)$

$=t^2+S^2+2t(S\cdot V)$

$2Pk=2\{(S+td)(S-C+mV)\}$

$2\{S^2-(S\cdot C)+m(S\cdot V)+t(d\cdot S)$

$-t(d\cdot C)+tm(d\cdot V)\}$

$2S^2-2(S\cdot C)+2m(S\cdot V)+2t(d\cdot S)-2t(d\cdot C)+2tm(d\cdot V)$

$k^2=(S-C+mV)(S-C+mV)=S^2-(S\cdot C)+m(S\cdot V)-(C\cdot S)+(C\cdot C)-m(C\cdot V)$

$+m(V\cdot S)-m(V\cdot C)+m^2(V\cdot V)=S^2-2(S\cdot C)+2m(S\cdot V)+(C\cdot C)$

$-2m(C\cdot V)+m^2$

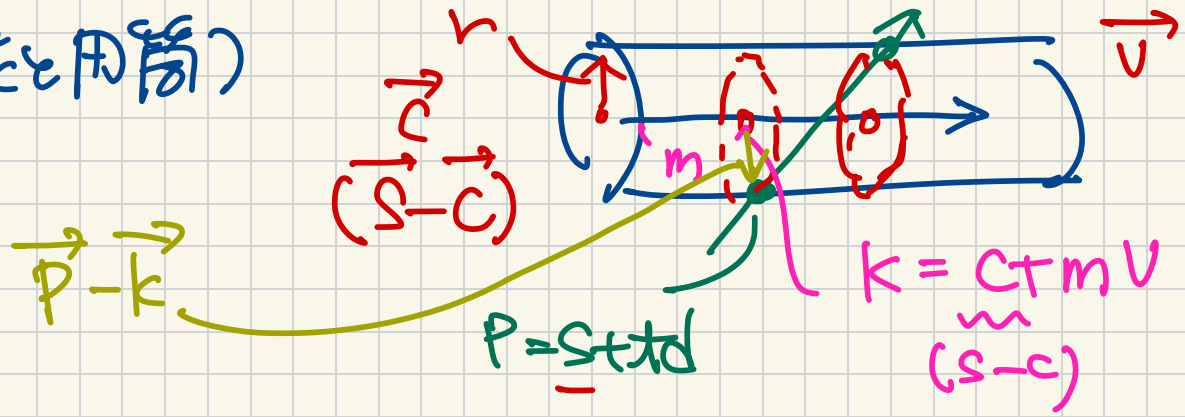
$m^2+t^2-S^2+2S^2+S^2+2t(S\cdot V)+2t(d\cdot S)$

$-2t(d\cdot C)+2tm(d\cdot V)+2m(S\cdot V)$

$+2m(S\cdot V)-2m(C\cdot V)+(C\cdot C)$

$-2(S\cdot C)-2(S\cdot C)-r^2$

[光線と円筒]



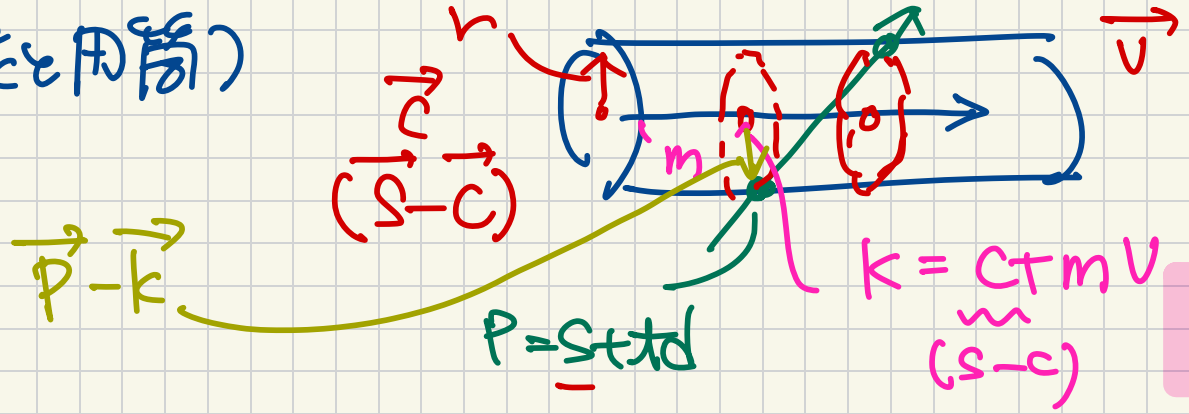
$\vec{p}=\vec{s}+t\vec{d}$

$\vec{k}=\vec{s}-\vec{c}+m\vec{v}$

$m^2+t^2+2(s\circ s)+2t(s\circ v)+$
$2t(d\circ s)-2t(d\circ c)+2tm(d\circ v)$
$+2m(s\circ v)+(c\circ c)-r^2=0$

$m^2+t^2-s^2+2s^2-s^2+2t(s\circ v)+2t(d\circ s)$
$-2t(d\circ c)+2tm(d\circ v)+2m(s\circ v)$
$+2m(s\circ v)-2m(c\circ v)+(c\circ c)$
$-2(s\circ c)-2(s\circ c)-r^2$
$-4(s\circ c)$

線と円筒



$\vec{p}=\vec{s}+t\vec{d}$

$\vec{k}=\vec{s}-\vec{c}+m\vec{v}$

$m=t(d\cdot v)+(c\cdot v)$

$m^2+t^2+2(s\cdot s)+2t(s\cdot v)+2t(d\cdot s)-2t(d\cdot c)+2tm(d\cdot v)$

$+2m(s\cdot v)+(c\cdot c)-r^2=0$

$2m(s\cdot v)=$

$m^2=\left((t(d\cdot v)+(c\cdot v))(t(d\cdot v)+(c\cdot v)\right)$

$2((t(d\cdot v)+(c\cdot v))(s\cdot v)$

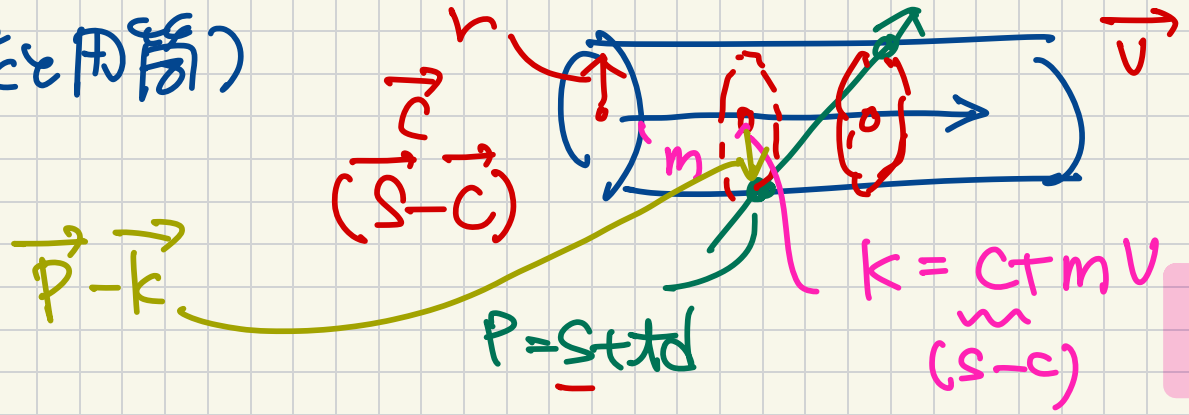
$=t^2(d\cdot v)^2+(c\cdot v)^2+2t(d\cdot v)(c\cdot v)$

$2t(d\cdot v)(s\cdot v)+2(c\cdot v)(s\cdot v)$

$2tm(d\cdot v)=2t(t(d\cdot v)+(c\cdot v))(d\cdot v)$

$2t^2(d\cdot v)^2+2t(c\cdot v)(d\cdot v)$

線と円筒



$\vec{p}=\vec{s}+t\vec{d}$

$\vec{k}=\vec{s}-\vec{c}+m\vec{v}$

$m=t(d\cdot v)+(c\cdot v)$

$m^2+t^2+2(s\cdot s)+2t(s\cdot v)+2t(d\cdot s)-2t(d\cdot c)+2tm(d\cdot v)$

$+2m(s\cdot v)+(c\cdot c)-r^2=0$

$2m(s\cdot v)=$

$m^2=\left((t(d\cdot v)+(c\cdot v)\right)\left(t(d\cdot v)+(c\cdot v)\right)$

$2\left(t(d\cdot v)+(c\cdot v)\right)(s\cdot v)$

$=t^2(d\cdot v)^2+(c\cdot v)^2+2t(d\cdot v)(c\cdot v)$

$2t(d\cdot v)(s\cdot v)+2(c\cdot v)(s\cdot v)$

$2tm(d\cdot v)=2t\left(t(d\cdot v)+(c\cdot v)\right)(d\cdot v)$

$2t^2(d\cdot v)^2+2t(c\cdot v)(d\cdot v)$

$t=\dfrac{-b\pm\sqrt{b^2-ac}}{a}$

$t^2\left(2(d\cdot v)^2+(d\cdot v)^2\right)$

$2t\left((s\cdot v)+(d\cdot s)-(d\cdot c)+2(d\cdot v)(c\cdot v)+(d\cdot v)(s\cdot v)\right)$

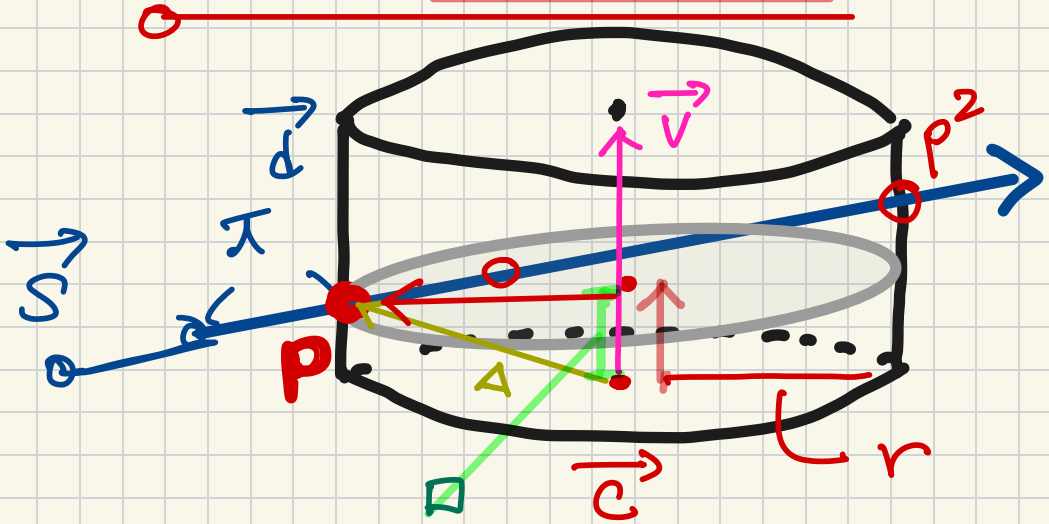
$2(s\cdot s)+(c\cdot c)-r^2+(c\cdot v)^2+2(c\cdot v)(s\cdot v)$

[シリンダーの接点]

レイ $\vec{P} = \vec{S} + t\vec{d}$

円筒 $\left[(P-C) - \{V\cdot(P-C)V\}\right]^2 = r^2$

$\vec{V}$ は単位方向
円筒の法線



$V\cdot(P-C)$ はスカラー長さ (mパラム) × $\vec{V}$

= P接点と Cの軸との最接近点

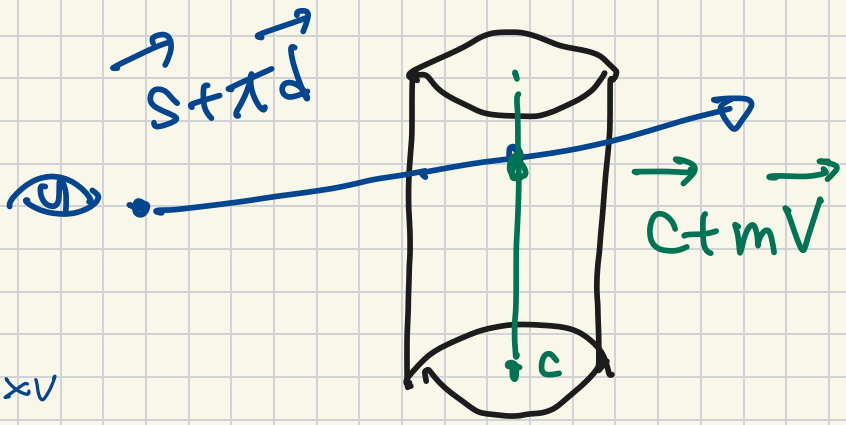
$$\left[S+td-C-\{V\cdot(S+td-C)V\}\right]^2 = r^2$$

$$a = d\cdot d - (V\cdot d)^2$$

$$b = 2\left[(d\cdot(S-C)) - (V\cdot d)(V\cdot(S-C))\right]$$

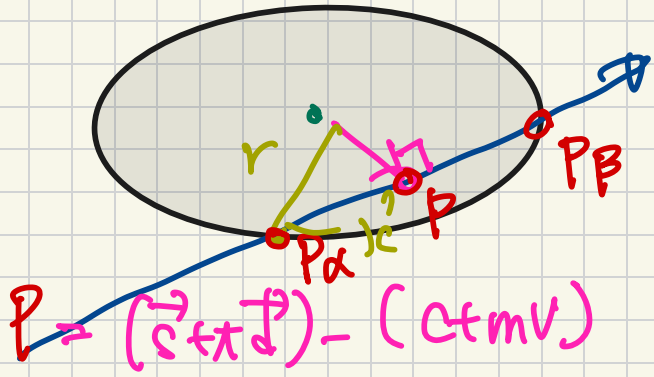
$$c = (S-C)\cdot(S-C) - (V\cdot(S-C))^2 - r^2$$

[線と線の接点]



① $S+\pi d$　② $C+mV$

$$S+\pi d = C+mV$$
$$\pi d = C+mV-S$$
$$\pi d \times V = (C+mV-S)\times V$$
$$\pi(d\times V) = m(V\times V) + (C-S)\times V$$
$$\pi(d\times V) = (C-S)\times V$$
$$\pi(d\times V)\circ(d\times V) = ((C-S)\times V)\circ(d\times V))$$
$$\pi = \frac{((C-S)\times V)\circ(d\times V))}{(d\times V)\circ(d\times V)}$$
$$\pi = \frac{((C-S)\times V)\circ(d\times V))}{\|d\times V\|^2}$$



$P = (\vec{S}+\pi\vec{d}) - (C+mV)$

if($|P| \leq r$)

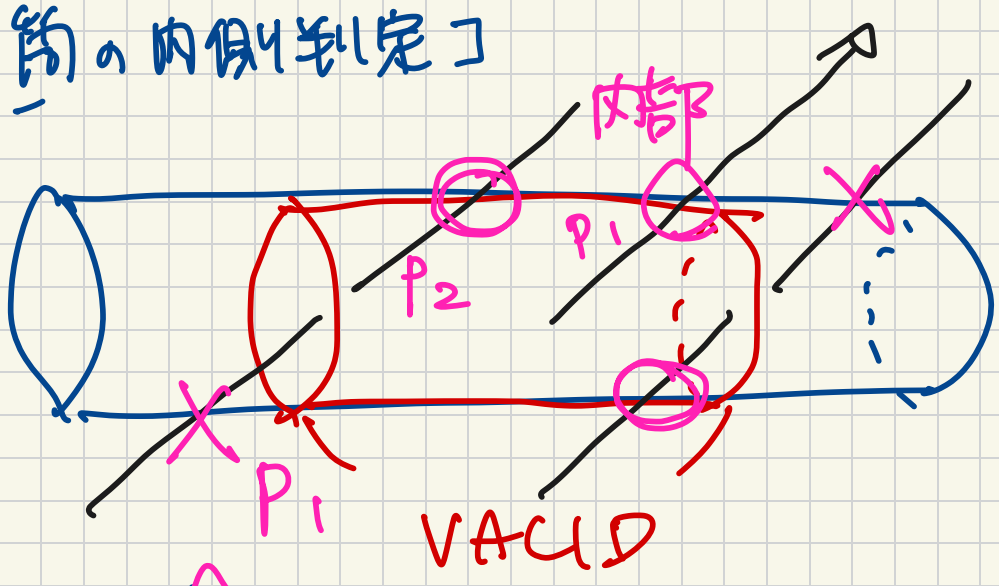
$$P_\alpha = \vec{S}+\pi-x\vec{d}$$
$$P_\beta = \vec{S}+\pi+x\vec{d}$$

$$\begin{cases} r^2 = x^2 + |P|^2 \\ x^2 = |P|^2 - r^2 \\ x = \sqrt{|P|^2 - r^2} \end{cases}$$

[法線ベクトル]

$$r = P_\alpha - (C+mV)$$

[円筒の内側判定]



① if P1 && P2 > 0

if (P1 ≦ P2)

if (P1 != VACUD)

if (P2 == VACUD)

== 内部

② if (P1 && P1 VAC(P)

③ if (P2 && P2 VAC(P)

< Phone の反射モデル >

環境光 (光源からどこかで反射)
(ambient)

$$Ra = ka \times Ia$$

ka = 反射係数
Ia = 強度

拡散反射
(Diffuse)

$$Rd = \text{Intensity} \times kd(\hat{n} \cdot \hat{l})$$

kd = 拡散反射係数

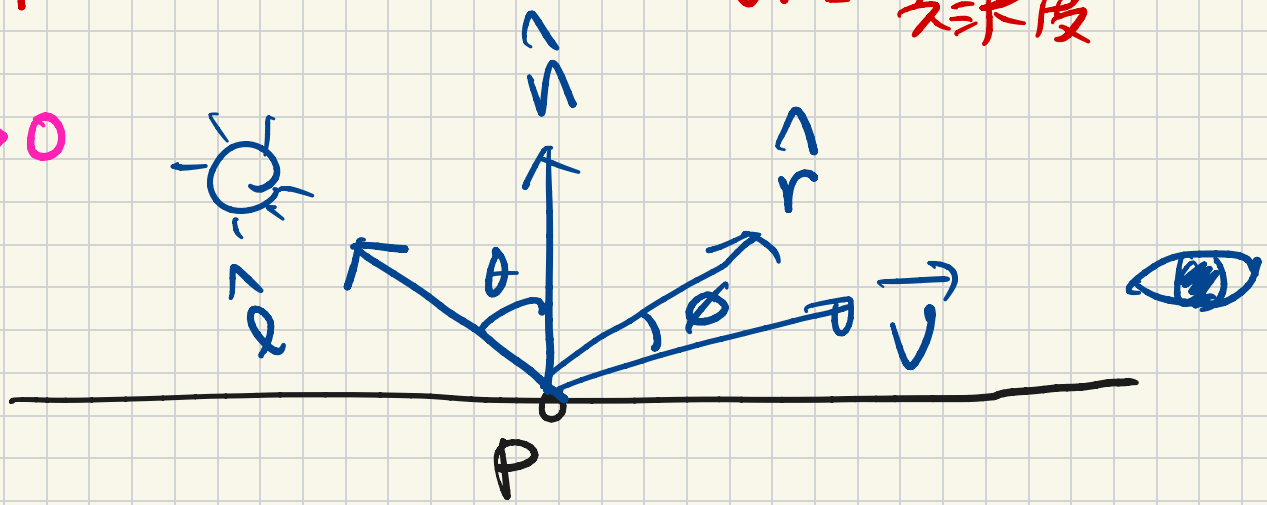
鏡面反射
(Specular)

$$Rs = \text{Intensity} \times ks(\hat{r} \cdot \hat{v})^{\alpha}$$

ks = 鏡面反射係数
α = 光沢度

Phong =

$$Ra + \sum_{light} \begin{cases} Rd + Rs & \hat{n} \cdot \hat{l} > 0 \\ 0 \end{cases}$$



拡散反射光 (diffuse)

$Rd = kd\ Ii\ (\vec{n} \cdot \vec{l})$

視点位置は関係なし

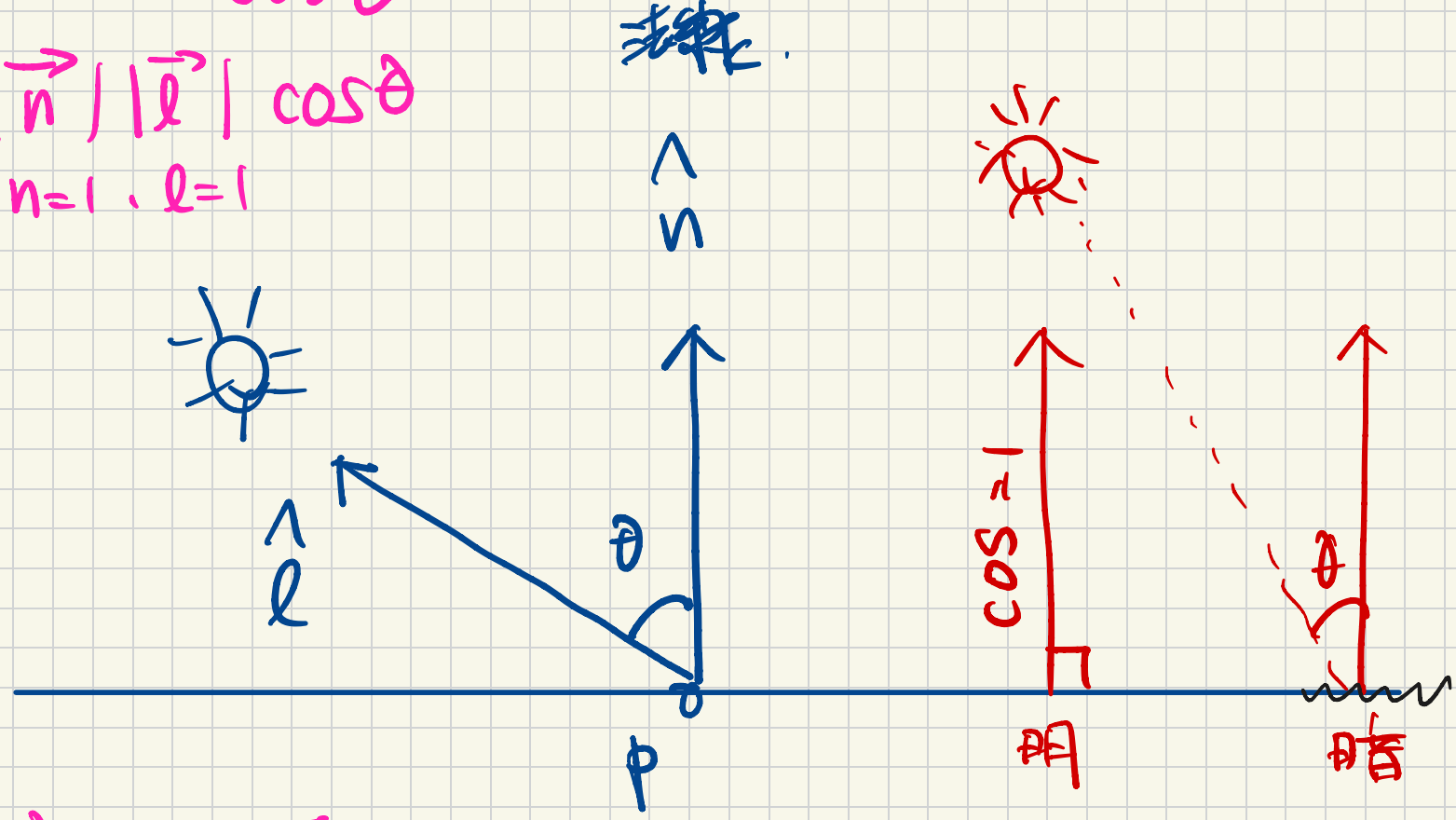
kd = 反射係数

Ii = 強度

$\cos\theta$

$\vec{n} \cdot \vec{l} = |\vec{n}||\vec{l}|\cos\theta$

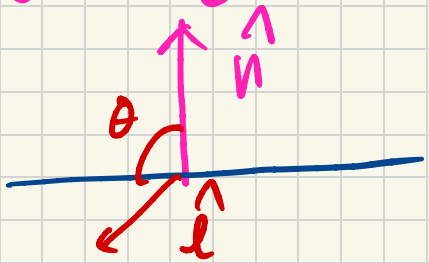
$n=1,\ l=1$



* $(n \cdot l) < 0$ の時は zero

$\theta > 90°$

面の裏側



鏡面反射：

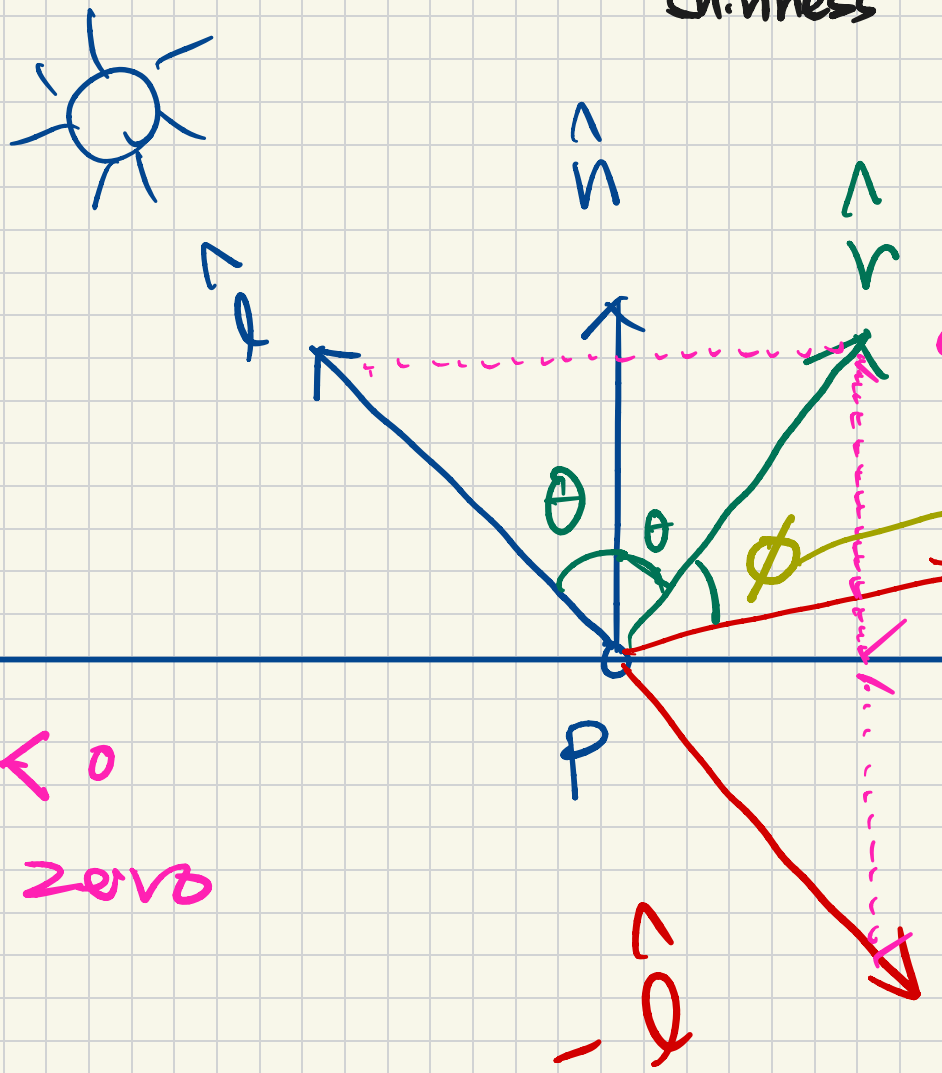
$$R_s = k_s I_i \cos^{\alpha}\phi$$

$$= k_s I_i (\vec{v}\cdot\vec{r})^{\alpha}$$

視点の位置は関係なし

k_s 鏡面反射係数
I_i 入射光の強度
$\vec{v}$ = 視線ベクトルの逆ベクトル
$\vec{r}$ = 入射光の正反射ベクトル
ϕ = $\vec{v}$ と $\vec{r}$ の対角
α = 光沢度 shinness

α = 光沢度 shinness



$r = 2(n\cdot l)n - l$

$\cos\theta\,\vec{n}$
$= (\vec{n}\cdot\vec{l})\vec{n}$

$\cos\theta$

$\vec{n}\cdot\vec{l} = |\vec{n}||\vec{l}|\cos\theta$

単位ベクトルは1

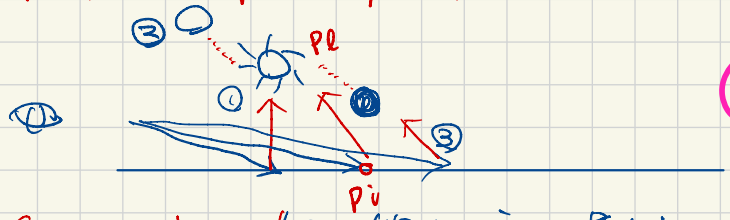
$\vec{v}$
(=-d)
2x

$\vec{v}\cdot\vec{r}$

$\vec{v}\cdot\vec{r} = |\vec{v}||\vec{r}|\cdot\cos\theta$
1 1

$*(\vec{v}\cdot\vec{r}) < 0$
n 時は zero

[陰影処理]
光源と交点の間に物体がある？



① シャドウレイの方向に物体なし = 影なし
② シャドウレイの方向に物体あり、但し光源より遠い = 影なし
③ シャドウレイの方向に物体あり = 光源を無視

↳ if（シャドウレイの方向に物体 純 |物体| < |光源|）

点光源：シャドウレイの長さ D_l
$D_l = |P_l - P_i|$ ⇒ 正規化前の $\vec{l}$ の長さ

平行光源：光源のpositionのなし = $D_l = \infty$ 無限遠方

Distance light
while (shapes)

① if (shadow ray と shapes intersect)

かつ

② (Distance shape < Distance light)

→ diffuse と specular を 0 として計算。

※シャドウレイの始点

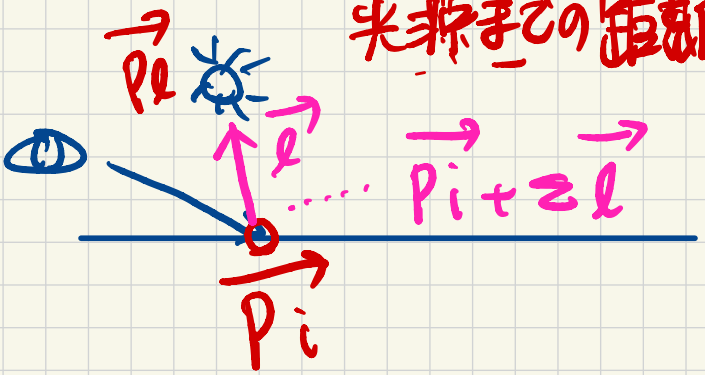
視線と物体の交点、とすると、シャドウレイは物体自身と交差してしまう！
⇒ ほんの少しずらす

シャドウレイ：始点ベクトル $\vec{P_i} + \varepsilon\vec{l}$

ε = 定数 1.0f / 512

方向ベクトル $\vec{l}$ （入射ベクトル = 交差 → 光源）

光源までの距離



$D'_l = D_l - \varepsilon$

$D_l = |P_{light} - P_{intersect}|$

$\vec{l} = \vec{P_l} - \vec{P_i}$ の正規化

問題点：平面の下から見た時に影が映る

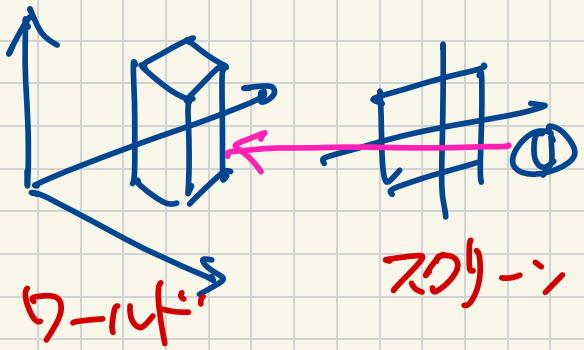
→ 影を消す = 光がシースルーしない
= 真黒にする？

→ 影のまま = 物体はシースルー ✓ ここを将来

[カメラ] 座標変換

① ワールド座標：ゲーム内の座標

② スクリーン座標：視線ベクトル



→ 接点の計算 = スクリーン ⇒ ワールドに変換

スクリーン座標の x、y、z をワールドのそれに合せる

[ワールド座標 ⇒ スクリーン座標]

① スクリーンの原点をワールド座標に平行移動

② y軸まわりに α の回転 ⇒ z軸を y-z 平面

③ x軸まわりに β の回転 ⇒ z軸を z と一致

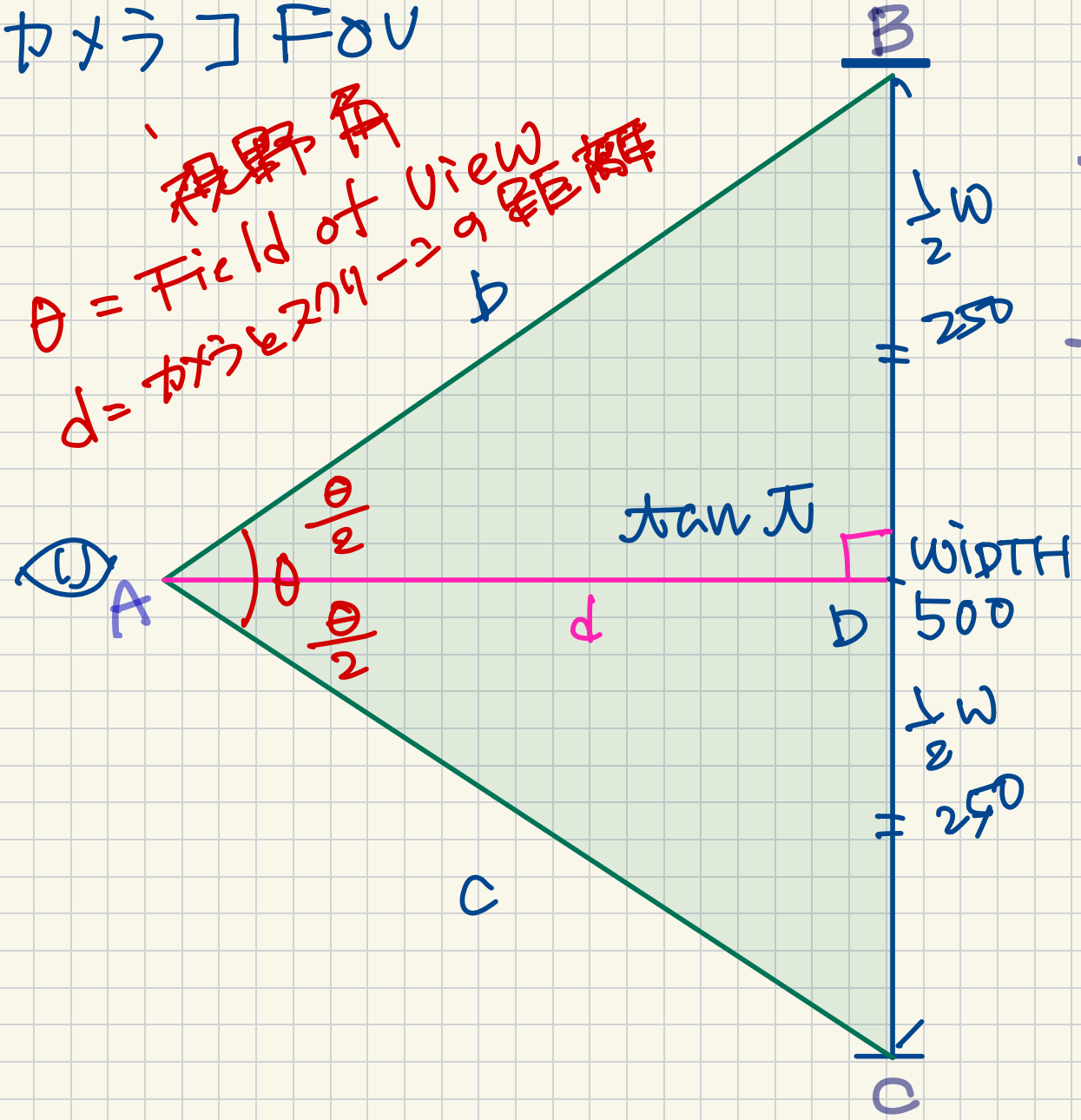
④ z軸まわりに γ の回転 ⇒ x'軸を x と一致(y も)

[スクリーン座標 ⇒ ワールド座標 は 4、3、2、1 と逆]

【カメラ】FoV

θ = Field of View（視野角）

d = カメラとスクリーンの距離



d は？

$$\frac{\frac{1}{2}W}{d} = \tan\left(\frac{1}{2}\theta\right)$$

$$\frac{1}{d} = \frac{\tan\left(\frac{1}{2}\theta\right)}{\frac{1}{2}W}$$

$$d = \frac{\frac{1}{2}W}{\tan\left(\frac{1}{2}\theta\right)}$$

$$d = \frac{W}{2} \times \frac{1}{\tan\left(\frac{1}{2}\theta\right)}$$

① スクリーン原点 ⇐ ワールド座標 $(\vec{P})$

$\Rightarrow$ スクリーン座標のワールド空間の原点の位置

$$\vec{P'} = \vec{P} - \vec{O}$$

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} - \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix} = \begin{bmatrix} P_x - O_x \\ P_y - O_y \\ P_z - O_z \end{bmatrix}$$

(+)

② y軸まわりに α 回転

$$\cos\alpha = \frac{P_z}{\sqrt{P_x^2 + P_z^2}}$$

$$\sin\alpha = \frac{P_x}{\sqrt{P_x^2 + P_z^2}}$$

次の計算に必要

③ x軸まわりに β 回転

$$\sin\beta = P_y$$

$$x = \sqrt{P_x^2 + P_z^2}$$

$$\cos\beta = x$$

④ z軸まわりに γ 回転

① $\vec{原点'}$ = ワールド座標の中での スクリーン座標の原点の位置

$\vec{原点'} = \vec{s} + t\vec{d}$

$\vec{s}$ = eye position

$\vec{d} = |\vec{0} - \vec{s}|$

目標点 $0 = (0, 0, 0)$

$t = \frac{w}{2} \times \frac{1}{\tan(\frac{1}{2}\theta)}$ FoV

$\vec{p'} = \vec{p} - \vec{原点'}$ vec3 (x, y, z)

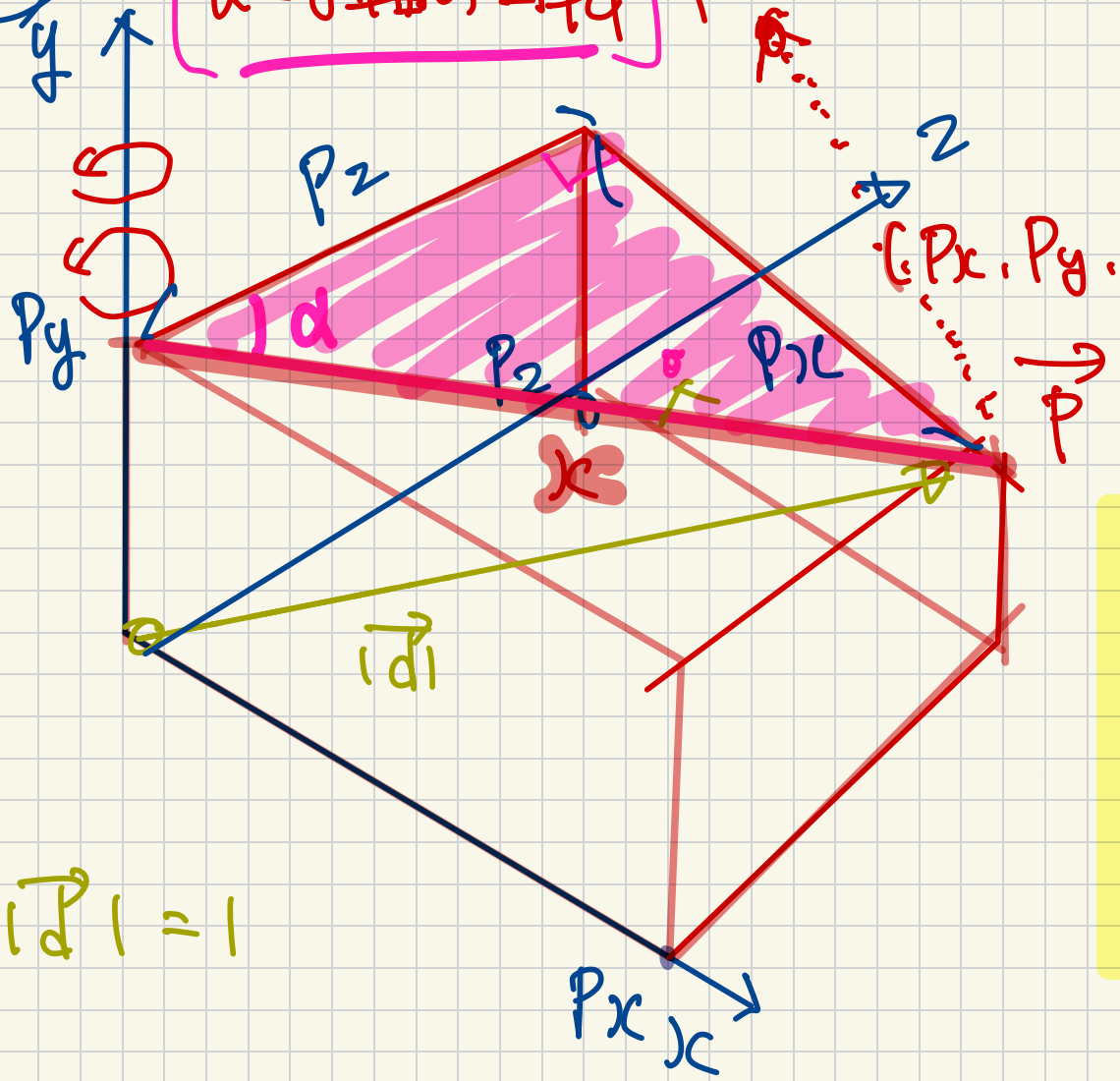
$\vec{p'}$ ワールド座標

$\vec{p}$ スクリーン座標

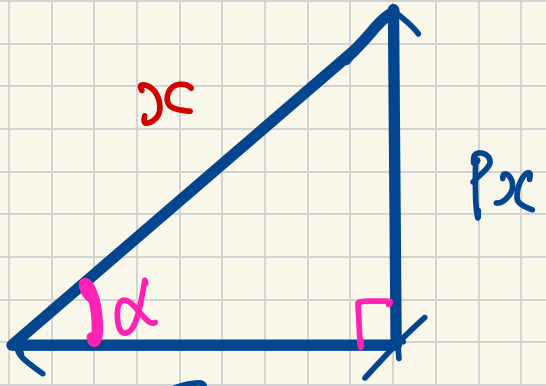
$\vec{原点'}$ ワールド座標の中での基準点

$\vec{p} = \vec{p'} + \vec{原点'}$

② [α=y軸の回転] P'



α

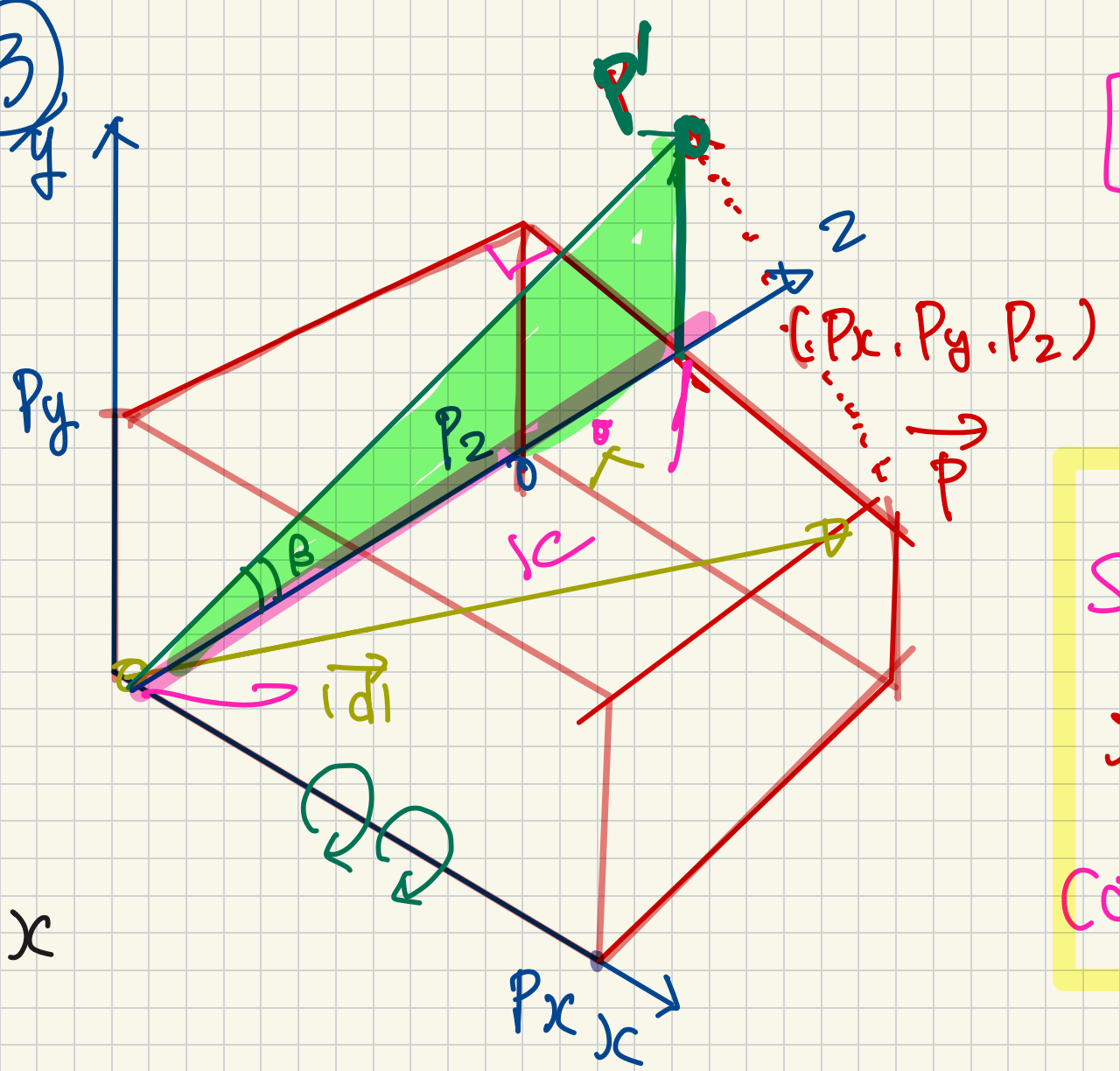


$$x = \frac{P_z}{\sqrt{P_{xc}^2 + P_z^2}}$$

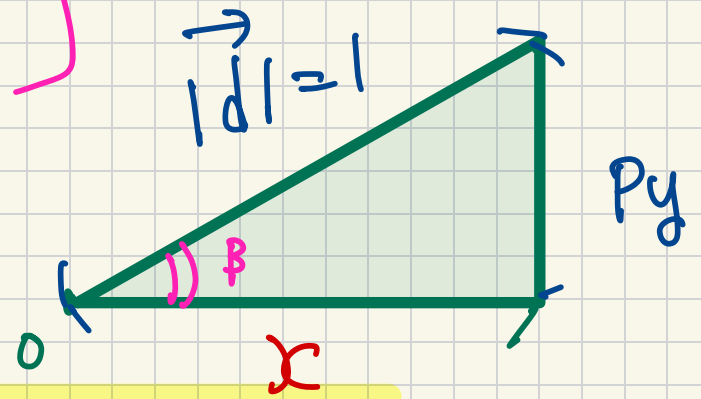
$$\cos\alpha = \frac{P_z}{\sqrt{P_{xc}^2 + P_z^2}}$$

$$\sin\alpha = \frac{P_{xc}}{\sqrt{P_{xc}^2 + P_z^2}}$$

③



[β]



$$\sin\beta = \frac{Py}{yc}$$

$$yc = \sqrt{Px^2 + P_2^2}$$

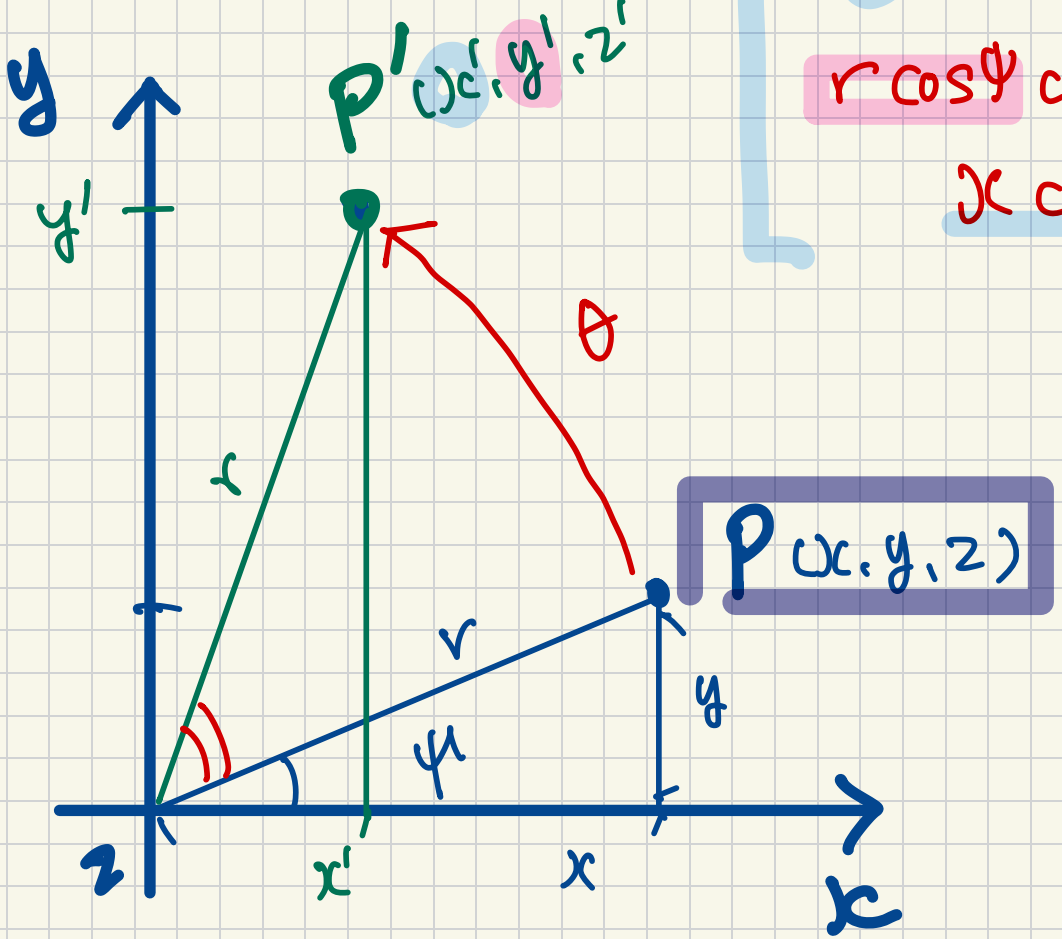
$$\cos\beta = \frac{yc}{}$$

β = x軸の回転

「回転の操作」　点 $P(x,y,z) \to$ 点 $P'(x',y',z')$

角度θを z軸回転.

① z座標は変らず。



$x = r\cos\psi$, $y = r\sin\psi$

$x' = r\cos(\psi+\theta)$

$r\cos\psi\cos\theta - r\sin\psi\sin\theta$

$x\cos\theta - y\sin\theta$

加法定理

加法定理

$\cos(A+B) = \cos A\cos B - \sin A\sin B$

$\sin(A+B) = \sin A\cos B + \cos A\sin B$

$y' = r\sin(\psi+\theta)$

加法定理

$r\sin\psi\cos\theta + r\cos\psi\sin\theta$

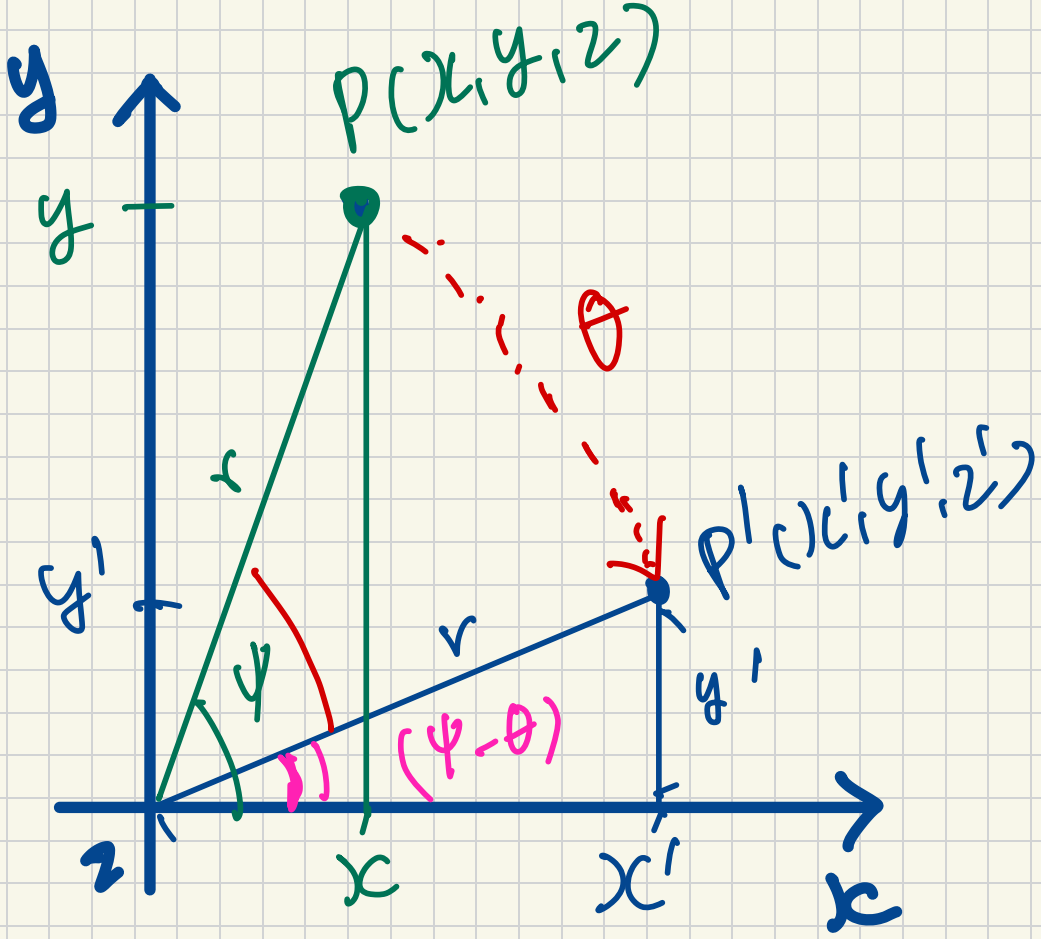
$y\cos\theta + x\sin\theta$

［回転の操作］

① z座標は変らず。

点 $P(x,y,z) \to$ 点 $P'(x',y',z')$

角度θを z軸回転.



$x = r\cos\psi$, $y = r\sin\psi$

$x' = r\cos(\psi-\theta)$

$r\cos\psi\cos\theta + r\sin\psi\sin\theta$

$x\cos\theta + y\sin\theta$

~~加法定理~~

$\cos(A-B) = \cos A\cos B + \sin A\sin B$

$\sin(A-B) = \sin A\cos B - \cos A\sin B$

$y = r\sin(\psi-\theta)$

$r\sin\psi\cos\theta - r\cos\psi\sin\theta$

$y\cos\theta - x\sin\theta$

[回転の操作 cont.]

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

$\Rightarrow$ 行列

[z軸回り] $R_z(\theta)$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

[x軸まわり] $R_x(\theta) \Leftarrow R_x(\beta)$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$R_z(\gamma)$

[y軸まわり] $R_y(\theta) \Leftarrow R_y(-\alpha)$ ∴ y軸の回転は負の方向

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

[座標変換その2]

$$\overrightarrow{Ray} = \overrightarrow{S} + t\overrightarrow{d}$$

$$\overrightarrow{S} = \overrightarrow{P}_{camera}$$

$$\overrightarrow{d} = \frac{\overrightarrow{dsc} + \overrightarrow{P_S}}{|\overrightarrow{dsc} + \overrightarrow{P_S}|}$$

$\overrightarrow{dsc}$ = カメラからスクリーン中心へのベクトル

$\overrightarrow{P_S}$ = スクリーン上のポイント

(カメラ→スクリーン→ワールド)

$$\overrightarrow{dsc} = \overrightarrow{d}_{camera} \circ \left(\frac{w}{2} \times \frac{1}{\tan(\frac{\theta}{2})}\right)$$

カメラの方向ベクトル

$$\overrightarrow{P_S} = x' = x \circ \overrightarrow{esx}$$
$$y' = y \circ \overrightarrow{esy}$$

$$\hat{k} = \frac{target - eye}{|target - eye|}$$

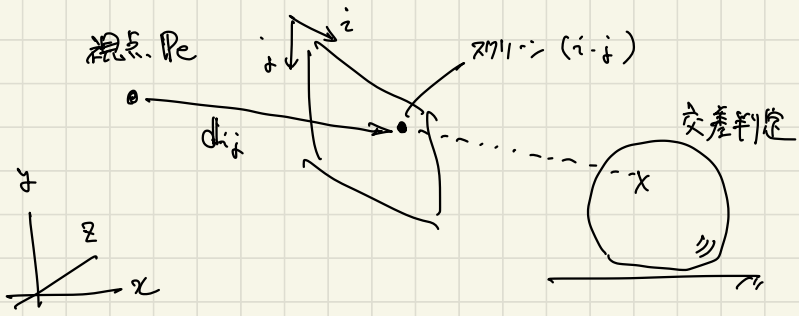
$$\hat{i} = up \times \hat{k}$$

$$\hat{j} = \hat{k} \times \hat{i}$$

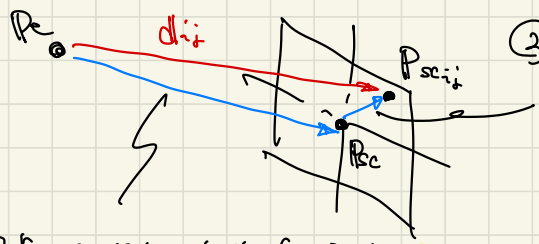
カメラ（スクリーン）　（左きき）　[takuraIno/-t]

レイと物体の交差判定のため、スクリーン上の点 (i,j) に向かう視線ベクトル d_{ij} を求めたい。

given：視点 P_e、視線の方向 d_e、視野角 FOV、スクリーンの大きさ $(H \times W)$



d_{ij} を求めるには、以下 2 つのベクトルが分かればよい。



② スクリーン中心から
スクリーン上の点 (i,j) に向かうベクトル

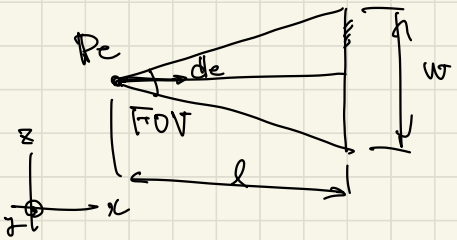
① 視点からスクリーン中心に向かうベクトル

① 視点からスクリーン中心に向かうベクトル

視野角 FOV，視線の方向 d_e，スクリーンの幅 W から求めることができる．

P_e からスクリーン中心までの距離 l は，

$$l = \frac{\frac{W}{2}}{\tan\left(\frac{FOV}{2}\right)}$$



なので，求めるベクトルは，

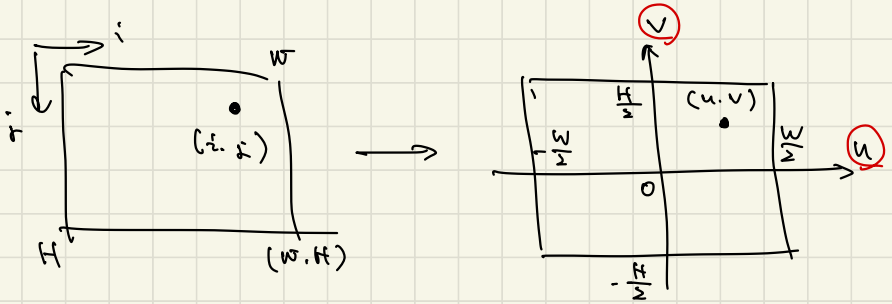
$l\, d_e$

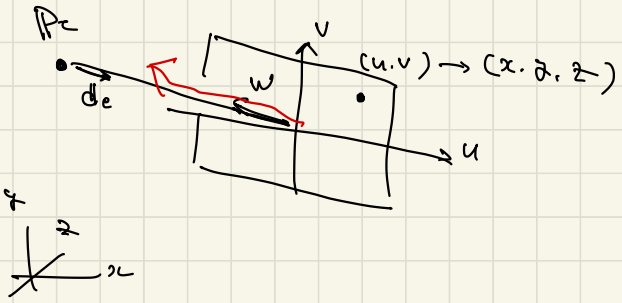
である //

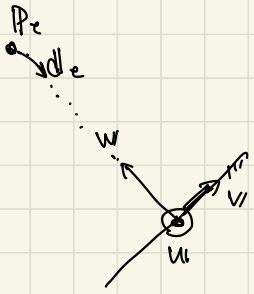
② スクリーン中心からスクリーン上の点 (i, j) に向かうベクトル

方向ベクトル d_e のスクリーン上の点 i, j $(0 \le i < W,\ 0 \le j < H)$ を

x, y, z 座標で表すため，スクリーン中心を原点とする (u, v, w) 座標系と考える．







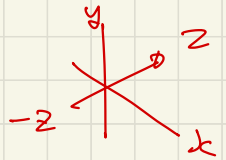
スクリーンの中心に (u, w, v) を考える。 uv空間の基底ベクトル u, v, w は、

forward: $w = \frac{de}{|de|}$ W

right: $u = w \times \frac{y}{|y|}$ W

up: $v = u \times w$

① なぜマイナス



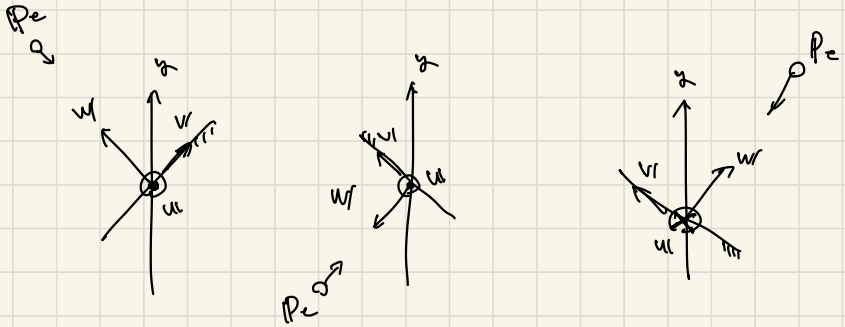
③ y ベクトルは $(0, 1, 0)$ でよい

ここで、$w = \pm y$ のときには、$u = 0$ になってしまうため、以下のように おこなう 必要がある。

日比先生の外積のやつ

$w = y$ のとき、 $u = x$, $v = z$.

$w = -y$ のとき、 $u = x$, $v = -z$



よって、ワールド (x, y, z) → スクリーン (u, w, v) への変換は、

x → x'

$$x' = T_r x$$

$$T_r = \begin{bmatrix} u' \\ w' \\ v' \end{bmatrix}$$

逆変換は、 $x = T_r^{-1} x'$

$= T_r^T x'$ で求められる。 //

流れ

1) (i, j) を (u, v) に変換

i	$0 \to W$
u	$-\frac{W}{2} \to \frac{W}{2}$

j	$0 \to H$
v	$\frac{H}{2} \to -\frac{H}{2}$

2) $(u, v) \to (u, w, v)$ とおき、ベクトルに、$(w = 0)$

$$x' = \begin{bmatrix} u \\ w \\ v \end{bmatrix} = \begin{bmatrix} u \\ 0 \\ v \end{bmatrix} \leftarrow 0$$

スクリーン座標？

3) T_r を求めておき、x' をワールド座標 x に変換

$$x = T_r x'$$

4) x' は uv 空間の原点基準なので、スクリーン中心からのベクトル。

すなわち、求める d_{ij} は、

center までの？

$$d_{ij} = l d_e + x' \quad \cdots (4-1)$$

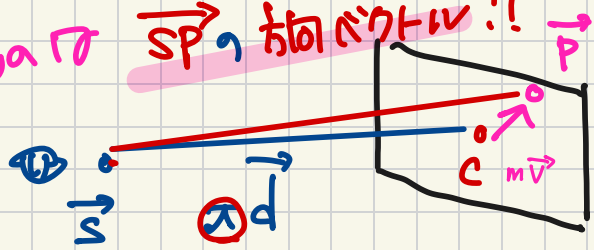
である。 [カメラの center]

* ベクトルを4次元に拡張すると、
(4-1)式は変換行列とベクトルの積のみで表現できる。

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} T_r & \vdots & l d_e \\ \cdots & \cdots & \cdots \\ 0 & \vdots & 1 \end{bmatrix} \begin{bmatrix} u \\ w \\ v \\ 1 \end{bmatrix}$$

[TAKCRAさんのを自分のに]

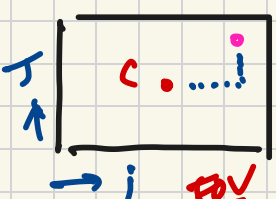
求めたいもの ⇒ $\overrightarrow{SP}$ の方向ベクトル!!



$C = \vec{S} + t\vec{d}$

$P' = (i, j, k)$

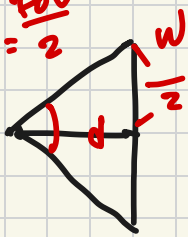
$P = P - \vec{C}$ にする



$t = \dfrac{w/2}{\tan(FoV/2)}$

ラジアンで

$\theta = \dfrac{FoV}{2}$



Ray-Tracing: Generating Camera Rays

Generating Camera Rays

Reading time: 22 mins.

Generating Camera Rays

First, let's recall that the purpose of a renderer is to assign a color to each pixel of the frame. The result must be an accurate representation of the scene geometry as seen from a particular 3D viewpoint. We also know from the previous chapter that parameters such as the field of view change how much of the scene we see. We have also explained in lesson 1 and in this lesson, that ray-traced images are created by generating one ray for each pixel in the frame. When a ray intersects an object from the scene, we set the pixel's color with the object's color at the intersection point. This process which we have already talked about in lesson 1 but will detail again in the next lessons, is known as **backward** or **eye-tracing** (because we follow the path of light rays from the camera to the object and from the object to the source, rather than from the light source to the object, and from the object to the camera).

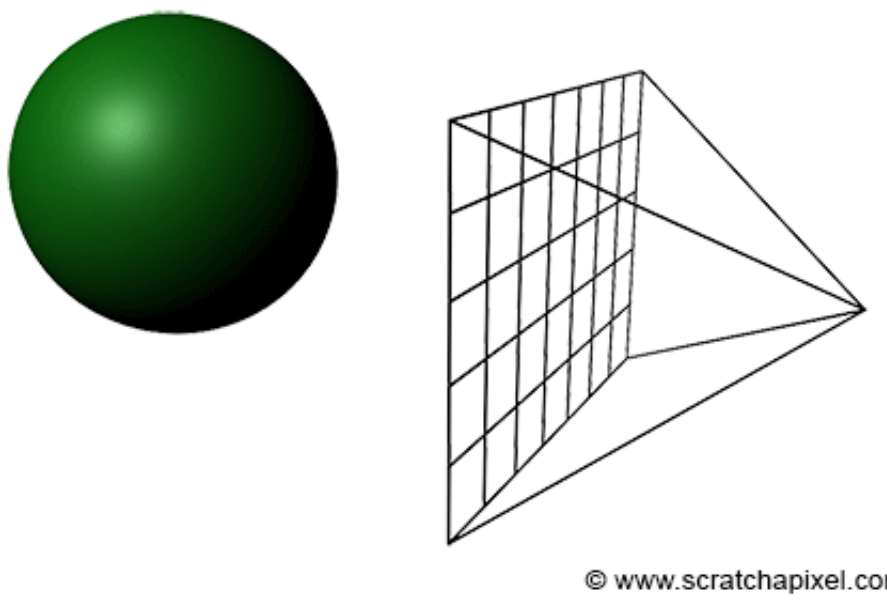


Figure 1: *backward or eye tracing consists of tracing rays from the eye through the center of each pixel of the image. If the ray intersects an object from the scene, the color of the pixel the ray is passing through is set with the color of the object at this intersection point.*

Naturally, the process of creating an image will start with constructing these rays which we call **primary** or **camera rays** (primary because these are the first rays we will cast into the scene. Secondary rays are shadows rays for example which we will talk about later). What do we know about these rays that would help us to construct them? We know that they start from the camera's origin. In almost all 3D applications, the default position of a camera when it is created is the origin of the world, which is defined by the point with coordinates (0, 0, 0). Remember from the lesson 3D Viewing: the Pinhole Camera Model, that the origin of the camera can be seen as the aperture of a pinhole camera (which is also the center of projection). The film of real-world pinhole cameras is located behind the aperture, which causes the light rays by geometrical construction to form an inverted image of the scene. However, this inversion can be avoided if the film plane lies on the

same side as the scene (in front of the aperture rather than behind). By convention, in ray tracing, it is often placed exactly 1 unit away from the camera's origin (this distance will never change and we will explain why further down). By convention, we will also orient the camera along the negative z-axis (the camera default orientation is left to the developer's choice, however generally the camera is oriented along either the positive or the negative z-axis. RenderMan, Maya, PBRT, and OpenGL align the camera along the negative z-axis and we suggest developers to follow the same convention). Finally, to make the beginning of the demonstration simpler, we will assume that our rendered image is square (the width and the height of the image in pixels are the same).

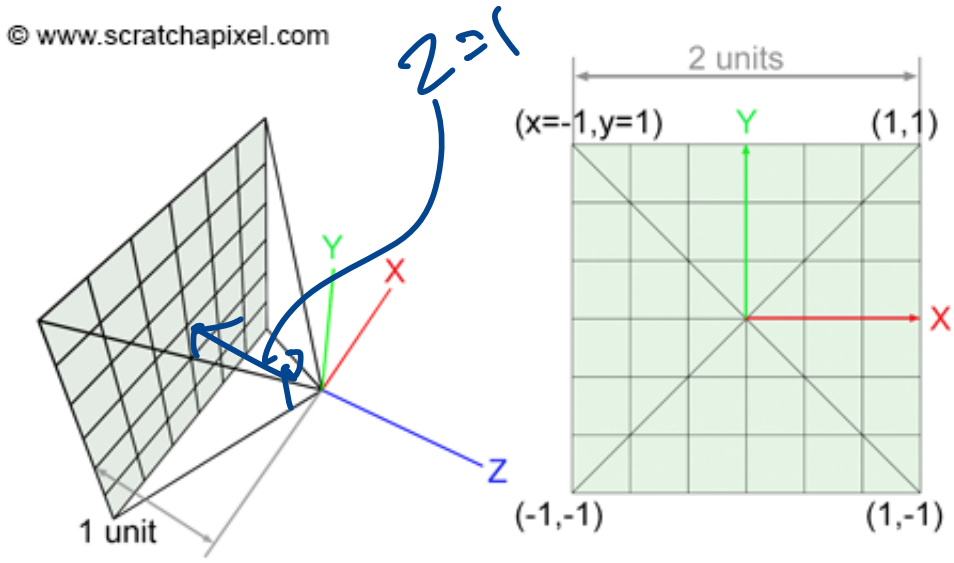


Figure 2: *Left: a basic camera. The original image size is 6x6 pixels and the eye's default position is the center of the world (0, 0, 0). Note how the camera points along the negative z-axis. The image plane is exactly 1 unit away from the origin. Right: pixels on the left of the y-axis and below the x-axis have negative world-space coordinates.*

Our task consists of creating a primary ray for each pixel of the frame. This can easily be done by tracing a line starting at the camera's origin and passing through the middle of each pixel (figure 1). We can express this line in the form of a ray whose origin is the camera's origin and whose direction is the vector from the camera's origin to the pixel center. To compute the position of a point at the center of a pixel, we need to convert the pixel coordinates which are originally expressed in **raster space** (the point coordinates are expressed in pixels with the coordinates (0,0) being the top-left corner of the frame) to **world space**.

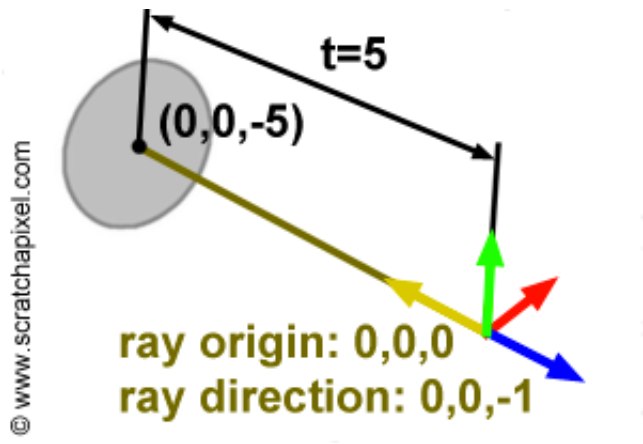
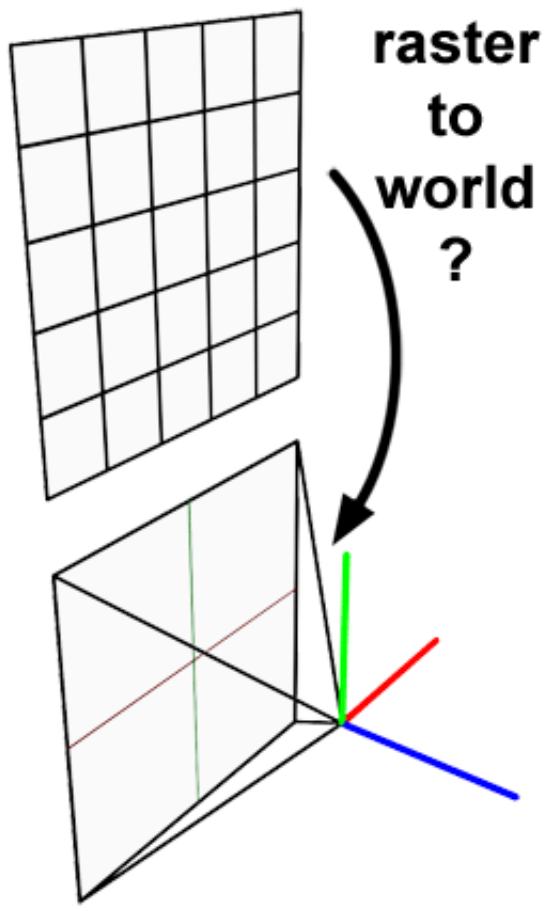


Figure 3: *ray-disk intersection.*



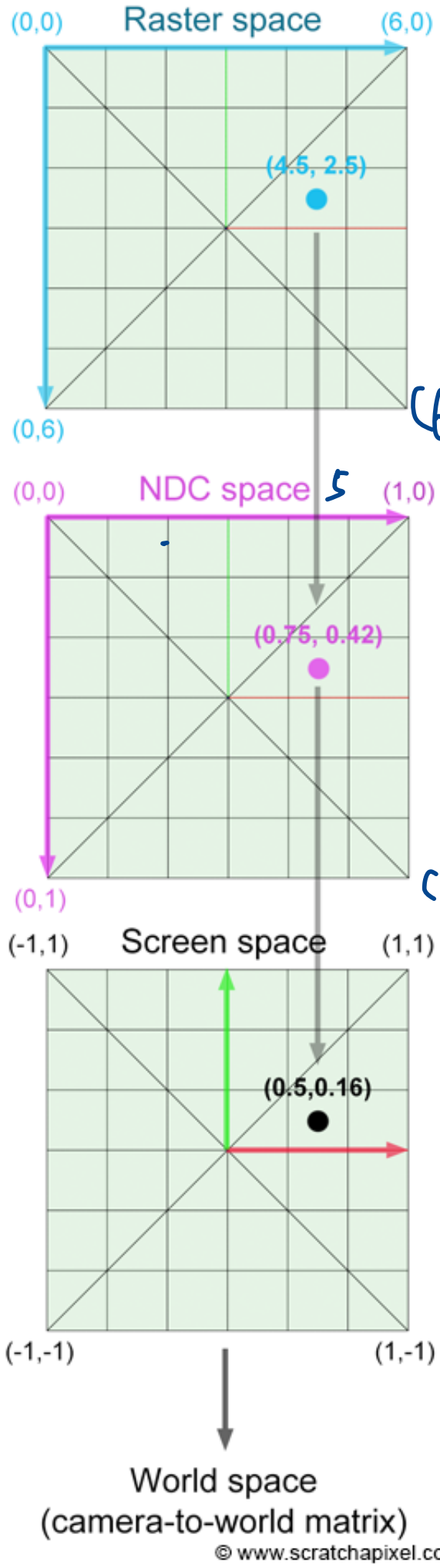
© www.scratchapixel.com

Figure 4: *from raster to screen-space.*

Why world space? World space is the space in which all objects of the scene, the geometry, the lights, and the cameras have their coordinates expressed into. For example, if a disk is located 5 units away from the world origin along the negative z-axis the world space coordinates of the disk are (0, 0, -5). If we want the mathematics for computing the intersection of a ray with this disk to work, the ray's origin and direction too, need to be defined in the same space. For example, if a ray has origin (0, 0, 0) and direction (0, 0, -1) where these numbers represent coordinates in the world space coordinate system, then the ray will intersect the disk at (0, 0, -5). This is shown in figure 3.

> ▼ 詳細
>
> You can find more information on spaces in the lesson Computing the Pixel Coordinates of a 3D point.

Another way of looking at the problem we are trying to solve is to start from the camera. We know the image plane is located exactly one unit away from the world origin and aligned along the negative z-axis. We also know that the image is square therefore the portion of the image plane on which the image is projected is also necessarily square. For reasons we will be explaining soon, the dimension of this projection area is 2 by 2 units (figure 2). We also know that a raster image is made of pixels. What we need is to find a relation between the coordinates of these pixels in raster space and the coordinates of the same pixels but expressed in world space. This process requires a few steps which are shown in figure 5.



pixelの解像度

(6.6)

[normalized device coord]

0~1に $\frac{x}{\text{pixelの数}}$ にconvert

(1.1)

-1~1 (2)のrange

Figure 5: *converting the coordinate of a point in the middle of a pixel to world coordinates requires a few steps. The coordinates of this point are first expressed in raster space (the pixels coordinate plus an offset of 0.5), then converted to NDC space (the coordinates are remapped to the range [0,1]) then converted to screen space (the NDC coordinates are remapped to the [-1,1]). Applying the final camera-to-world transformation 4x4 matrix transforms the coordinates in screen space to world space.*

We first need to normalize this pixel position using the frame's dimensions. The new normalized coordinates of the pixels are said to be defined in **NDC space** (which stands for Normalized Device Coordinates):

$$PixelNDC_x = \frac{(Pixel_x + 0.5)}{ImageWidth},$$
$$PixelNDC_y = \frac{(Pixel_y + 0.5)}{ImageHeight}.$$

Note that we add a small shift (0.5) to the pixel position because we want the final camera ray to pass through the middle of the pixel. Pixel coordinates expressed in NDC space are in the range [0,1] (yes, NDC space in ray tracing is different than NDC space in the rasterization world where it generally maps to the range [-1,1]). As you can see though in figure 2, the film or image plane is centered around the world's origin. In other words, pixels located on the left of the image should have negative x-coordinates, while those located on the right should have positive x-coordinates. The same logic applies to the y-axis. Pixels located above the line defined by the x-axis should have positive y-coordinates, while those located below should have negative y-coordinates. We can correct for this by remapping our normalized pixel coordinates which are currently in the range [0:1] to the range [-1:1]:

$$PixelScreen_x = 2 * PixelNDC_x - 1,$$
$$PixelScreen_y = 2 * PixelNDC_y - 1.$$

However notice that with this equation, $Pixel\ Remapped_y$ is negative for pixels located above the x-axis and positive for pixels located below (while it should be the other way around). The following formula will correct this problem:

$$PixelScreen_y = 1 - 2 * PixelNDC_y.$$

The value now varies from 1 to -1 as $Pixel_y$ varies from 0 to $ImageWidth$. Coordinates expressed in this manner are said to be defined in **screen space**.

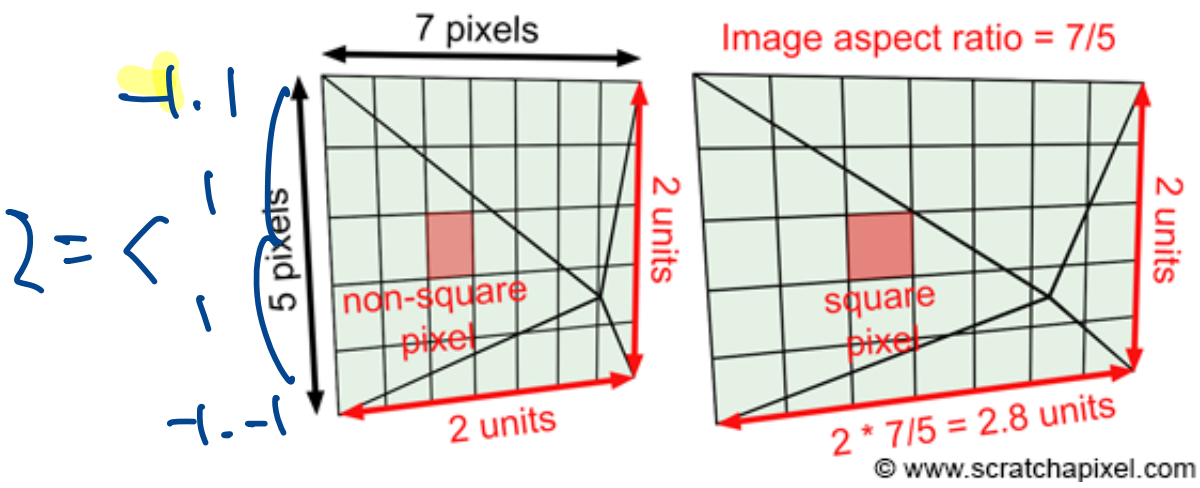


Figure 6. Left: *because the width and the height of the image are different, the pixels are not squared. To correct for this we need to scale the image plane along the x-axis by the image aspect ratio which can be computed from divided the width by the height (in pixels) of the image.*

Until now though we have assumed that the image was square. Accounting for the image aspect ratio is quite simple. Let's now look at a case where the image has dimensions 7 by 5 pixels (it is a small image but an image nonetheless). Dividing the width by the height of the image gives the value 1.4\. When the pixel coordinates are defined in screen space they are in the range [-1, 1]. However, there are more pixels along the x-axis (7) than there are along the y-axis (5) therefore the pixels are squashed and elongated along the vertical axis (see figure 6). To make them square again (as pixels should be) we need to multiply the pixels' x-coordinates by the image aspect ratio, which in this case, is 1.4 (see again figure 6). Note that this operation leaves the y- pixel coordinates (in screen space) unchanged. They are still in the range [-1,1] but the x pixel coordinates are now in the range [-1.4, 1.4] (are more generally [-aspect ratio, aspect ratio]).

$$ImageAspectRatio = \frac{ImageWidth}{ImageHeight},$$
$$PixelCamera_x = (2 * PixelScreen_x - 1) * ImageAspectRatio,$$
$$PixelCamera_y = (1 - 2 * PixelScreen_y).$$

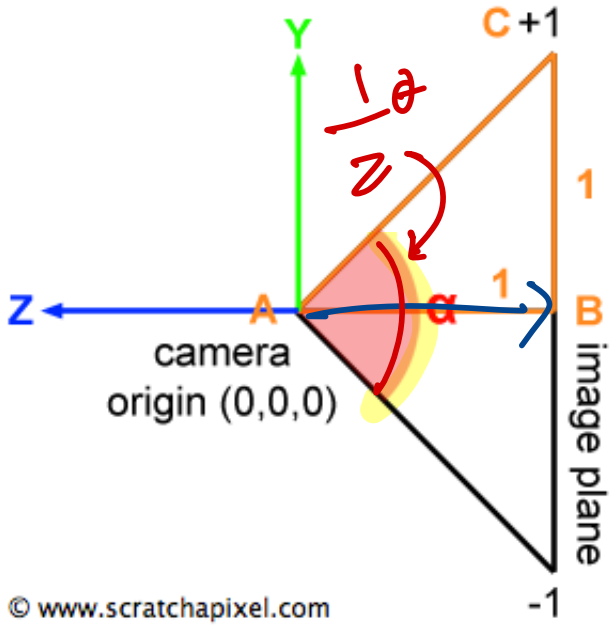


Figure 7: *a side view of our camera setup. The distance from the eye position to the image plane is 1 unit (vector AB). The distance from B to C is 1 unit as well. Using simple trigonometry we can then easily compute the angle α.*

Finally, we need to account for the field of view. Note that so far, the y coordinates of any point defined in screen space are in the range [-1, 1]. We also know that the image plane is 1 unit away from the camera's origin. If we look at the camera setup from the side view, we can draw a triangle by joining the camera's origin to the top and bottom edges of the film plane. Because we know the distance from the camera's origin to the film plane (1 unit) and the height of the film plane (2 units since it goes from y=1 to y=-1) we can use some simple trigonometry to find the angle of the right triangle ABC which is half of the vertical angle α (alpha), the angle we are interested in:

$$\frac{\alpha}{2} = atan(\frac{OppositeSide}{AdjacentSide}) = atan(\frac{1}{1}) = \frac{\pi}{4}.$$

In other words, the field of view or the angle α in the particular case is 90 degrees. Now notice that to compute the length of the line BC all we need is to compute the tangent of the angle α divided by 2:

$$|||BC|| = tan(\frac{\alpha}{2}).$$

Remember from the lesson on geometry that the notation ||V|| means the length of the vector V. We can also observe that for values of α greater than 90 degrees, ||BC|| is greater than 1, and for values lower than 90 degrees, ||BC|| is lower than 1\. For example if α=60, tan(60/2)=0.57 and if α =110, tan(110/2)=1.43\. We can therefore multiply the screen pixel coordinates (which at the moment are contained in the range [-1, 1]) by this number to scale them up or down. As you may have guessed, this operation changes how much of the scene we see, and is equivalent to zooming in (we see less of the scene when the field of view decreases) and out (and seeing more of the scene when the value of the field of view increases). In conclusion, we can define the field of view of the camera in terms of the angle α, and multiply the screen pixel coordinates with the result of the tangent of this angle divided by two (if this angle is expressed in degrees don't forget to convert it to radians):

$$PixelCamera_x = (2 * PixelScreen_x - 1) * ImageAspectRatio * tan(\frac{\alpha}{2}),$$
$$PixelCamera_y = (1 - 2 * PixelScreen_y) * tan(\frac{\alpha}{2}).$$

At this point, the original pixel coordinates are expressed with regard to the camera's image plane. They have been normalized, remapped between [-1:1], multiplied by the image aspect ratio, and multiplied by the tangent of the field of view angle α divided by 2\. This point is said to be in **camera space** because its coordinates are expressed with regard to the camera's coordinate system. When the camera is in its default position, the camera's coordinate system and the world's coordinate system are aligned. The point lies on the image plane which is 1 unit away from the

camera's origin but remembers that the camera is also aligned along the negative z-axis. Therefore we can express the final coordinate of the pixel on the image plane as:

$$P_{cameraSpace} = (PixelCamera_x, PixelCamera_y, -1)$$

This gives us the position P ($P_{cameraSpace}$) of a pixel in the image on the image plane of the camera. From there, we can compute a ray for this pixel by defining the origin of the ray as the camera's origin (let's call this point O) and the direction of the ray as the normalized vector OP (figure 8). The vector OP is simply the position of the point on the image plane minus the camera origin. The camera origin and the world Cartesian coordinate system are the same when the camera is in its default position, therefore the point O is simply (0, 0, 0).

In pseudo-code we get (you will find the actual C++ implementation at the end of the lesson):

```
float imageAspectRatio = imageWidth / (float)imageHeight; // assuming width > he
float Px = (2 * ((x + 0.5) / imageWidth) - 1) * tan(fov / 2 * M_PI / 180) * imag
float Py = (1 - 2 * ((y + 0.5) / imageHeight) * tan(fov / 2 * M_PI / 180);
Vec3f rayOrigin(0);
Vec3f rayDirection = Vec3f(Px, Py, -1) - rayOrigin; // note that this just equal
rayDirection = normalize(rayDirection); // it's a direction so don't forget to n
```

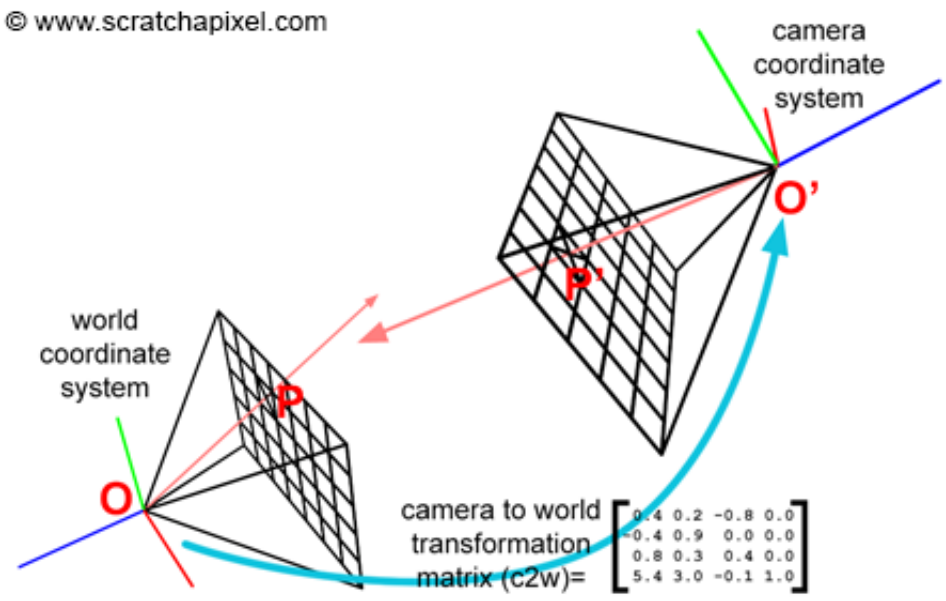


Figure 8: *we can move the camera in space to frame the scene as we want. The final position and orientation of the camera can be represented by a 4x4 matrix which we usually call the camera-to-world transformation matrix. If we know 0 (the origin of the camera which is also the origin of the world coordinate system) and P (the position in word space of the pixel the ray passes through) we can easily get 0' and P' by multiplying O and P by the camera-to-world camera matrix. Finally, the ray direction can be computed as P'-O'.*

Finally, we want to be able to render an image of the scene from any particular point of view. After you have moved the camera from its original position (centered at the origin of the world coordinate system and aligned along the negative z-axis) you can express the translation and rotation values of the camera with a 4x4 matrix. Usually, this matrix is called the **camera-to-world matrix** (and its inverse is called the **world-to-camera matrix**). If we apply this camera-to-world matrix to our points O and P then the vector ||O'P'|| (where O' is the point O and P' is the point P transformed by the camera-to-world matrix) represents the normalized direction of the ray in world space (figure 8). Applying the camera-to-world transform to O and P transforms these two points from camera space to world space. Another option is to compute the ray direction while the camera is in its default position (the vector OP), and apply the camera-to-world matrix to this vector.

Note how the camera coordinate system moves with the camera. Our pseudo code can easily be modified to account for camera transformation (rotation and translation, scaling a camera are not particularly recommended):

```
float imageAspectRatio = imageWidth / imageHeight; // assuming width > height
float Px = (2 * ((x + 0.5) / imageWidth) - 1) * tan(fov / 2 * M_PI / 180) * imag
float Py = (1 - 2 * ((y + 0.5) / imageHeight) * tan(fov / 2 * M_PI / 180);
Vec3f rayOrigin = Point3(0, 0, 0);
Matrix44f cameraToWorld;
cameraToWorld.set(...); // set matrix
Vec3f rayOriginWorld, rayPWorld;
cameraToWorld.multVectMatrix(rayOrigin, rayOriginWorld);
cameraToWorld.multVectMatrix(Vec3f(Px, Py, -1), rayPWorld);
Vec3f rayDirection = rayPWorld - rayOriginWorld;
rayDirection.normalize(); // it's a direction so don't forget to normalize
```

To compute the final image we will need to create a ray for each pixel of the frame using the method we have just described and test if any one of these rays intersects the geometry from the scene. Unfortunately, we are not to a point yet in this series of lessons where we can compute the intersection between rays and objects but this will be the topic of the next two lessons.

Source Code

The source code of this lesson is just a simple example of how the rays can be generated for each pixel of an image. The code loops over all the pixels of the image (line 13-14), and compute a ray for the current pixel. We have combined all the remapping steps described in this chapter in one single line of code. The original x-pixel coordinate is divided by the image width to remap the initial coordinate to the range [0,1]. The resulting value is then remapped to the range [-1,1], scaled by the `scale` variable (line 9) and the image aspect ratio (line 10). The pixel y-coordinate is transformed similarly, however remember that the y-normalized coordinate needs to be flipped (line 16). At the end of this process, we can create a vector using the transformed point x and y coordinates. The z coordinate of this vector is set to minus one (line 18): by default, the camera looks down the negative z-axis. The resulting vector is finally transformed by the camera-to-world camera and normalized. The camera's origin is also transformed (line 12) by the camera-to-world matrix. We can finally pass the ray direction and origin transformed to world space to the `rayCast` function.

```
void render(
    const Options &options,
    const std::vector<std::unique_ptr<Object>> &objects,
    const std::vector<std::unique_ptr<Light>> &lights)
{
    Matrix44f cameraToWorld;
    Vec3f *framebuffer = new Vec3f[options.width * options.height];
    Vec3f *pix = framebuffer;
    float scale = tan(deg2rad(options.fov * 0.5));
    float imageAspectRatio = options.width / (float)options.height;
    Vec3f orig;
    cameraToWorld.multVecMatrix(Vec3f(0), orig);
    for (uint32_t j = 0; j < options.height; ++j) {
        for (uint32_t i = 0; i < options.width; ++i) {
            float x = (2 * (i + 0.5) / (float)options.width - 1) * imageAspectRa
            float y = (1 - 2 * (j + 0.5) / (float)options.height) * scale;
            Vec3f dir;
            cameraToWorld.multDirMatrix(Vec3f(x, y, -1), dir);
            dir.normalize();
            *(pix++) = castRay(orig, dir, objects, lights, options, 0);
        }
    }

    // Save result to a PPM image (keep these flags if you compile under Windows
    std::ofstream ofs("./out.ppm", std::ios::out | std::ios::binary);
    ofs << "P6\n" << options.width << " " << options.height << "\n255\n";
    for (uint32_t i = 0; i < options.height * options.width; ++i) {
        char r = (char)(255 * clamp(0, 1, framebuffer[i].x));
```

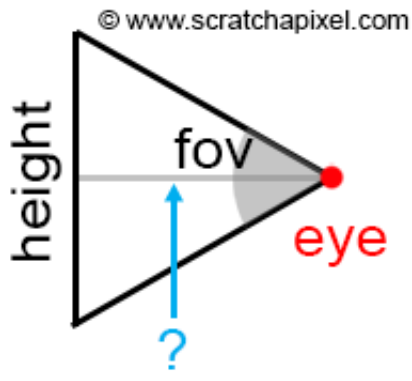
```
            char g = (char)(255 * clamp(0, 1, framebuffer[i].y));
            char b = (char)(255 * clamp(0, 1, framebuffer[i].z));
            ofs << r << g << b;
        }

        ofs.close();

        delete [] framebuffer;
    }
```

In the next lessons, we will show how we cast primary rays into the scene by calling the function `rayCast` which takes the ray origin and direction as argument (as well as other things such as a list of objects and lights, etc.) and returns a color. The function will return the color of the background if the ray didn't hit anything or the color of the object at the intersection point otherwise. Note that before we loop over all the pixels in the image to compute their color, we create a frame buffer in which we store the result of the `rayCast` function (lines 7). Once all the rays have been traced for all the pixels in the image we can then store the result of this image on disk. Unfortunately, we won't be able to implement the `rayCast` function until we get to the next lesson. In the meantime, we will be converting the ray direction into a color and store this color for the current pixel instead (lines 8-9 below). The final image is saved to disk in the ppm rabbits format (lines 25-34 above).

```
Vec3f castRay(
    const Vec3f &orig, const Vec3f &dir,
    const std::vector<std::unique_ptr<Object>> &objects,
    const std::vector<std::unique_ptr<Light>> &lights,
    const Options &options,
    uint32_t depth)
{
    Vec3f hitColor = (dir + Vec3f(1)) * 0.5;
    return hitColor;
}
```

▼ 詳細

In computer graphics, they are often different ways of getting the same result using different approaches. If you look at the source code of other render engines, you are likely to find that the problem of transforming rays from image space to world space can be solved in many different ways. However, the result should always be the same of course regardless of the approach taken.



For example, we can look at the problem in the following way: the pixel coordinates don't need to be normalized (in other words, converted from pixel coordinates to NDC and then screen space). We can compute the ray direction using the following equations:

$$\begin{aligned} d_x &= x - width/2 \\ d_y &= height/2 - y \\ d_z &= -(height/2)/tan(fov * 0.5). \end{aligned}$$

Where x and y are the pixel's coordinates, and fov is the vertical field-of-view. Keep in mind that d_z is negative because the camera by default is oriented along the negative z-axis. Then if we

normalize this vector, we get the same result as if we had been through a raster to NDC, and NDC-to-screen transform. If we transform this ray direction to world space (we multiply the vector d by the camera-to-world matrix), we get:

$$\begin{aligned} d_x &= (x - width/2)u_x + (height/2 - y)v_x - ((height/2)/tan(fov * 0.5))w_x \\ d_y &= (x - width/2)u_y + (height/2 - y)v_y - ((height.2)/tan(fov * 0.5))w_y \\ d_z &= (x - width/2)u_z + (height/2 - y)v_z - ((height/2)/tan(fov * 0.5))w_z \end{aligned}$$

If we expand and regroup the terms we get:

$$\begin{aligned} d_x &= xu_x - width/2u_x + height/2v_x - yv_x - (height/2)/tan(fov * 0.5)w_x \\ &= xu_x + y(-v_x) + (width/2u_x + height/2v_x - (height/2)/tan(fov * 0.5)w_x) \\ d_y &= xu_y - width/2u_y + height/2v_y - yv_y - (height/2)/tan(fov * 0.5)w_y \\ &= xu_y + y(-v_y) + (width/2u_y + height/2v_y - (height/2)/tan(fov * 0.5)w_y) \\ d_z &= xu_z - width/2u_z + height/2v_z - yv_z - (height/2)/tan(fov * 0.5)w_z \\ &= xu_z + y(-v_z) + (width/2u_z + height/2v_z - (height/2)/tan(fov * 0.5)w_z) \end{aligned}$$

Which we can re-write as (equation 1):

$$d = xu + y(-v) + w'.$$

Where $w' = (-width/2)u + (height/2)v - ((height/2)/tan(fov * 0.5))w.$

In other words, if we know the camera-to-world matrix, we can pre-compute the w' vector, compute the ray direction in word space using equation 1, and then normalize the resulting vector. In pseudo-code:

```
Vec3f w_p = (-width / 2 ) * u + (height / 2) * v - ((height / 2) / tan(fov_rad
Vec3f ray_dir = normalize(x * u + y * (-v) + w_p);
```

The vector w' only needs to be computed once and re-used each time we need to compute a new ray direction. The vectors u, v, and w are just the first, second, and third vectors of the camera-to-world matrix (the first three rows if you use row-major matrices).

Summary: We Are Almost Ready to Render our First Image

As we progress in this series of lessons, will be able to put all the different techniques we learned about so far to create a basic but functional ray-tracer (by functional we mean rendering some geometry and saving the resulting image to a file on disk). All we are now missing to get to that result is to learn about ray-geometry intersection which is the topic of the next lesson.

previous next

Computing the Pixel Coordinates of a 3D Point

Perspective Projection

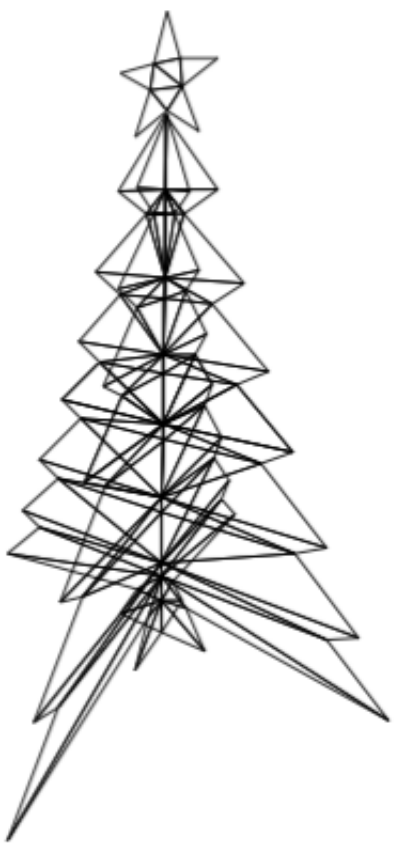
Reading time: 8 mins.

How Do I Find the 2D Pixel Coordinates of a 3D Point?

"**How do I find the 2D pixel coordinates of a 3D point?**" is one of the most common questions in 3D rendering on the Web. It is an essential question because it is the fundamental method to create an image of a 3D scene. In this lesson, we will use the term **rasterization** to describe the process of finding 2D pixel coordinates of 3D points. In its broader sense, Rasterization refers to converting 3D shapes into a raster image. A raster image, as explained in the previous lesson, is the technical term given to a digital image; it designates a two-dimensional array (or rectangular grid if you prefer) of pixels.

Don't be mistaken: different rendering techniques exist for producing images of 3D scenes. Rasterization is only one of them. Ray tracing is another. Note that all these techniques rely on the same concept to make that image: the idea of **perspective projection**. Therefore, for a given camera and a given 3D scene, all rendering techniques produce the same visual result; they use a different approach to produce that result.

Also, computing the 2D pixel coordinates of 3D points is only one of the two steps in creating a photo-realistic image. The other step is the process of shading, in which the color of these points will be computed to simulate the appearance of objects. You need more than just converting 3D points to pixel coordinates to produce a "complete" image.



To understand rasterization, you first need to be familiar with a series of essential techniques that we will also introduce in this chapter, such as:

- The concept of local vs. global coordinate system.
- Learning how to interpret 4x4 matrices as coordinate systems.

- Converting points from one coordinate system to another.

Read this lesson carefully, as it will provide you with the fundamental tools that almost all rendering techniques are built upon.

We will use matrices in this lesson, so read the Geometry lesson if you are uncomfortable with coordinate systems and matrices.

We will apply the techniques studied in this lesson to render a **wireframe** image of a 3D object (adjacent image). The files of this program can be found in the source code chapter of the lesson, as usual.

A Quick Refresher on the Perspective Projection Process

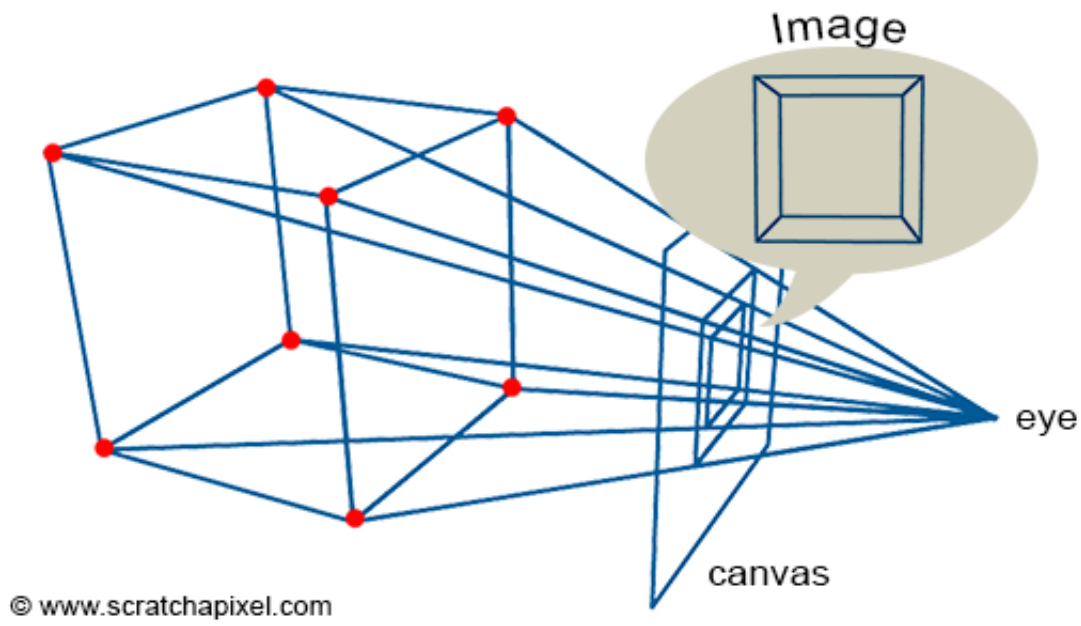


Figure 1: *to create an image of a cube, we need to extend lines from the corners of the object towards the eye and find the intersection of these lines with a flat surface (the canvas) perpendicular to the line of sight.*

We talked about the perspective projection process in quite a few lessons already. For instance, check out the chapter The Visibility Problem in the lesson "Rendering an Image of a 3D Scene: an Overview". However, let's quickly recall what perspective projection is. In short, this technique can be used to create a 2D image of a 3D scene by projecting points (or vertices) that make up the objects of that scene onto the surface of a canvas.

We use this technique because it is similar to how the human eye works. Since we are used to seeing the world through our eyes, it's pretty natural to think that images created with this technique will also look natural and "real" to us. You can think of the human eye as just a "point" in space (Figure 2) (of course, the eye is not exactly a point; it is an optical system converging rays onto a small surface - the retina). What we see of the world results from light rays (reflected by objects) traveling to the eye and entering the eye. So again, one way of making an image of a 3D scene in computer graphics (CG) is to do the same thing: project vertices onto the surface of the canvas (or screen) as if the rays were sliding along straight lines that connect the vertices to the eye.

It is essential to understand that perspective projection is just an arbitrary way of representing 3D geometry onto a two-dimensional surface. This method is most commonly used because it simulates one of the essential properties of human vision called **foreshortening**: objects far away from us appear smaller than objects close by. Nonetheless, as mentioned in the Wikipedia article on perspective), it is essential to understand that the perspective projection is only an **approximate representation** of what the eye sees, represented on a flat surface (such as paper). The important word here is "approximate".

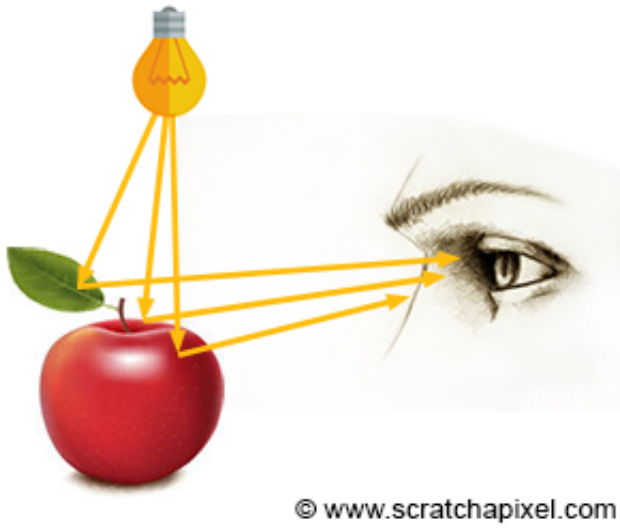


Figure 2: *among all light rays reflected by an object, some of these rays enter the eye, and the image we have of this object, is the result of these rays.*

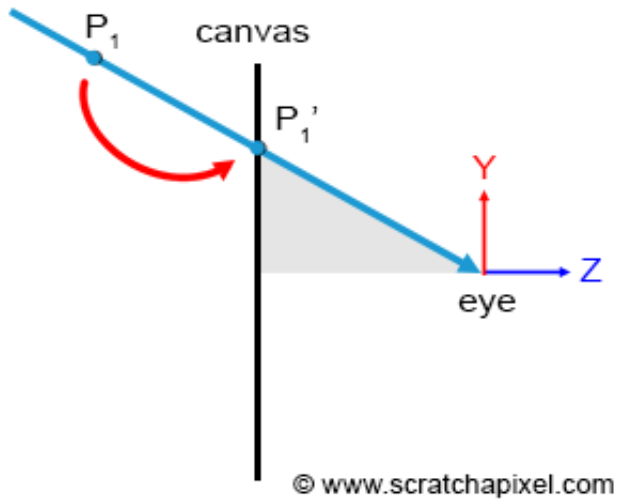


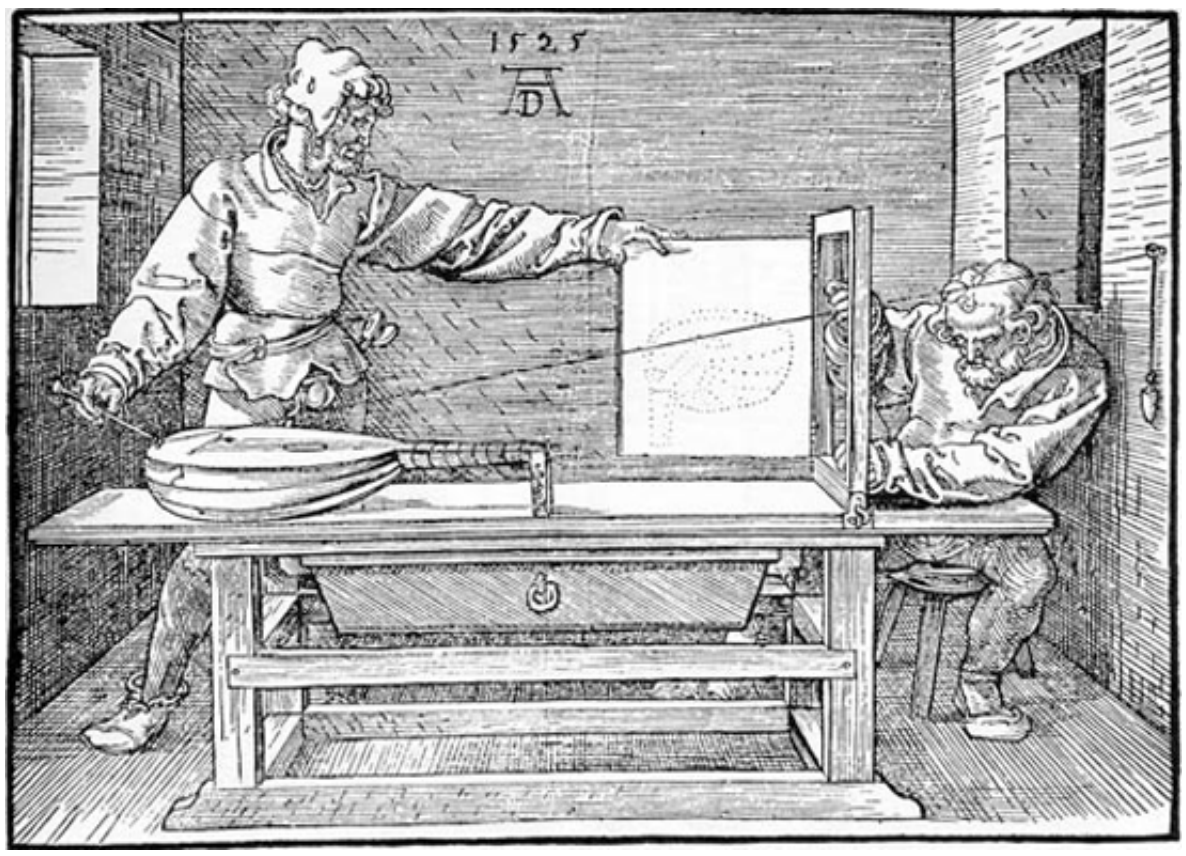
Figure 3: *we can think of the projection process as moving a point down along the line that connects the point to the eye. We can stop moving the point along that line when the point lies on the plane of the canvas. We don't explicitly "slide" the point along this line, but this is how the projection process can be interpreted.*

In the lesson mentioned above, we also explained how the world coordinates of a point located in front of the camera (and enclosed within the viewing frustum of the camera, thus visible to the camera) could be computed using a simple geometric construction based on one of the properties of similar triangles (Figure 3). We will review this technique one more time in this lesson. The equations to compute the coordinates of projected points can be conveniently expressed as a 4x4 matrix. The computation is simple but a series of operations on the original point's coordinates: this is what you will learn in this lesson. However, by expressing the computation as a matrix, you can reduce these operations to a single point-matrix multiplication. This approach's main advantage is representing this critical operation in such a compact and easy-to-use form. It turns out that the perspective projection process, and its associated equations, can be expressed in the form of a 4x4 matrix, as we will demonstrate in the lesson devoted to the the perspective and orthographic projection matrices. This is what we call the **perspective projection matrix**. Multiplying any point whose coordinates are expressed with respect to the **camera coordinate system** (see below) with this perspective projection matrix will give you the position (or coordinates) of that point on the canvas.

> ▼ 詳細
>
> In CG, transformations are almost always linear. But it is essential to know that the perspective projection, which belongs to the more generic family of **projective transformation**, is a non-linear transformation. If you're looking for a visual explanation of which transformations are linear and which transformations are not, this Youtube video does a good job.

Again, in this lesson, we will learn about computing the 2D pixel coordinates of a 3D point without using the perspective projection matrix. To do so, we will need to learn how to "project" a 3D point onto a 2D drawable surface (which we will call in this lesson a canvas) using some simple geometry rules. Once we understand the mathematics of this process (and all the other steps involved in computing these 2D coordinates), we will then be ready to study the construction and use of the perspective projection matrix: a matrix used to simplify the projection step (and the projection step only). This will be the topic of the next lesson.

Some History



The mathematics behind perspective projection started to be understood and mastered by artists towards the end of the fourteenth century and the beginning of the fifteenth century. Artists significantly contributed to educating others about the mathematical basis of perspective drawing through books they wrote and illustrated themselves. A notable example is "The Painter's Manual" published by Albrecht Dürer in 1538 (the illustration above comes from this book). Two concepts broadly characterize perspective drawing:

- Objects appear smaller as their distances to the viewer increase.
- **Foreshortening**: the impression, or optical illusion, that an object or a distance is smaller than it is due to being angled towards the viewer.

Another rule in foreshortening states that vertical lines are parallel, while nonvertical lines converge to a perspective point, appearing shorter than they are. These effects give a sense of depth, which helps evaluate the distance of objects from the viewer. Today, the same mathematical principles are used in computer graphics to create a perspective view of a 3D scene.

-

next

Placing a Camera: the LookAt Function

Framing: The LookAt Function

Reading time: 12 mins.

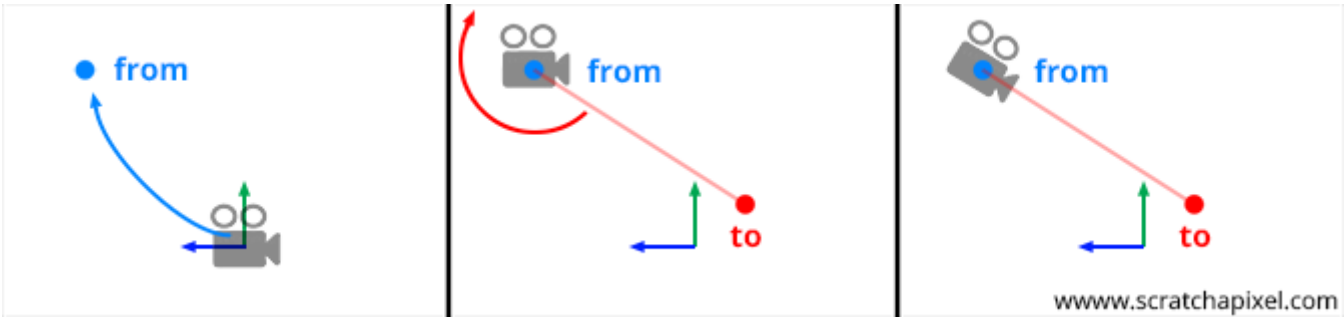
▼ Details

In this short lesson, we will study a simple but useful method to place 3D cameras. To understand this lesson, you will need to be familiar with the concept of transformation matrix and cross-product between vectors. If that's not already the case, you might want to read the lesson Geometry first.

Placing the Camera

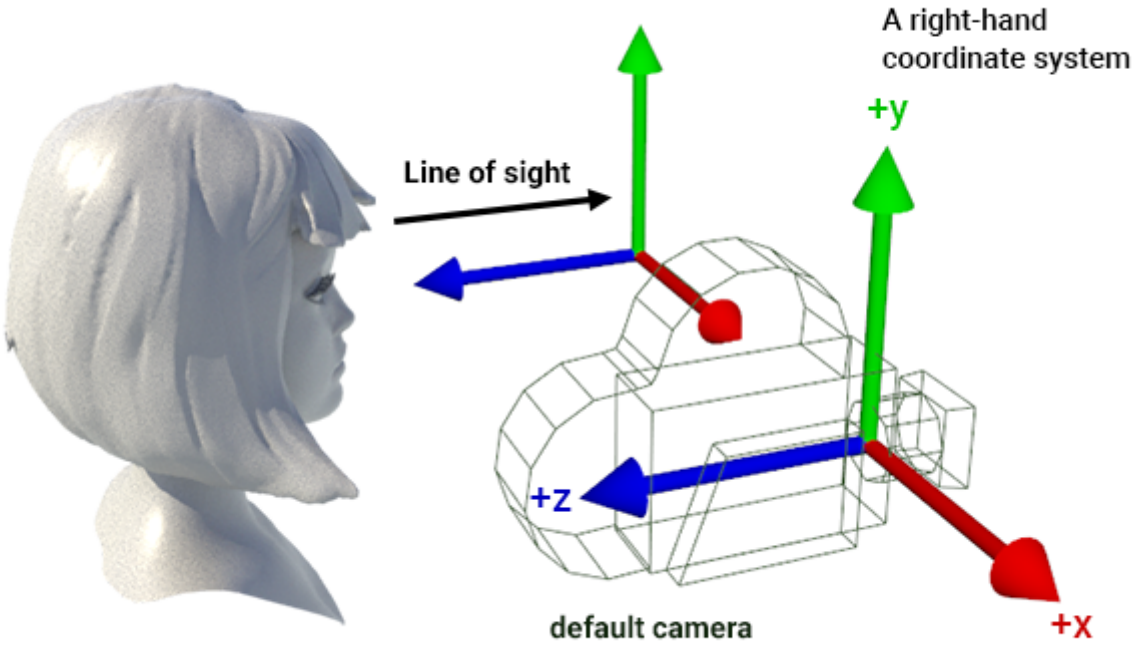
Being able to place the camera in a 3D scene is essential. However, in most of the lessons from Scratchapixel, we usually set the camera position and rotation (remember that scaling a camera doesn't make sense) using a 4x4 matrix which is often labeled the **camera-to-world** matrix. However, setting up a 4x4 matrix by hand is not friendly.

Thankfully, we can use a method that is commonly referred to as the **look-at** method. The idea is simple. To set a camera position and orientation, all you need is a point in space to set the camera position and a point to define what the camera is looking at (an aim). Let's label our first point "from" and our second point "to".



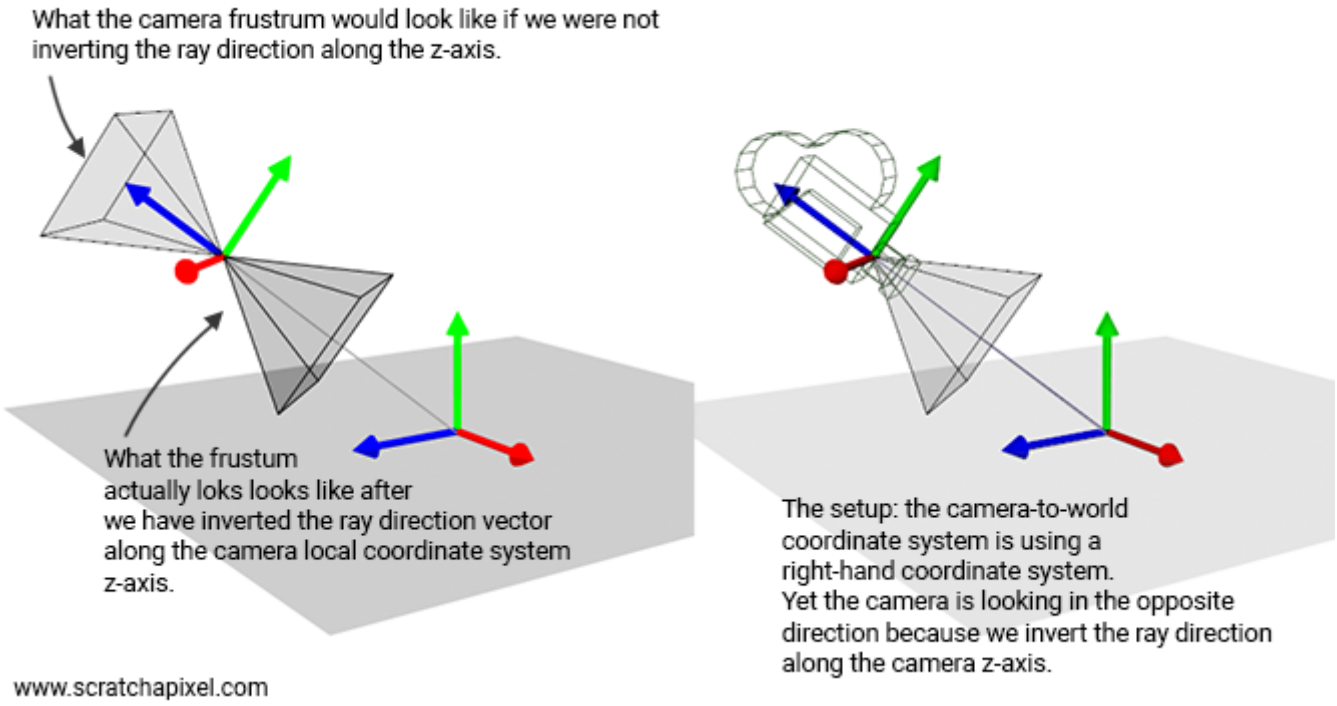
We can easily create a world-to-camera 4x4 matrix from these two points as we will demonstrate in this lesson.

Before we get any further, however, let's address an issue that can be a source of confusion. Remember that in a right-hand coordinate system, if you are looking along the z-axis, the x-axis is pointing to the right, the y-axis is pointing upward and the z-axis is pointing towards you as shown in the figure below.



Therefore quite naturally, when we think of creating a new camera, it feels normal to orient the camera as if we were looking at the right-hand coordinate system with the z-axis pointing towards the camera (as shown in the image above). Because by convention cameras are oriented that way, books (e.g. Physically Based Rendering / PBRT) sometimes suggest that this is because cameras are not defined in a right-hand coordinate system but a left-hand one. If you look down the z-axis, a left-hand coordinate system is one in which the z-axis points away from you (in the same direction as the line of sight). Assuming the right-hand coordinate is the rule, why should we make an exception for cameras? This explanation is not inaccurate as such, it is nonetheless potentially a source of confusion.

We prefer to say that cameras are using a right-hand coordinate system like all the other objects in our 3D application. However, we do flip the orientation of the camera at render time, by "scaling" the ray direction by -1 along the camera's local coordinate z-axis when we cast rays into the scene. If you check the lesson Ray-Tracing: Generating Camera Rays you will notice that the ray-direction z-component is set to -1 before the ray direction vector is itself transformed by the camera-to-world matrix. This is not *stricto sensu* a scaling. We just flip the direction of the ray direction vector along the camera's local coordinate system z-axis.



Bottom line: if you use a right-hand coordinate system for your application, to keep things consistent, the camera should also be defined in a right-hand coordinate system like with any other 3D object. But as we cast rays in the opposite direction, it is as if the camera was indeed looking down along the negative z-axis. With this clarification out of the way, let's now see how we build this matrix.

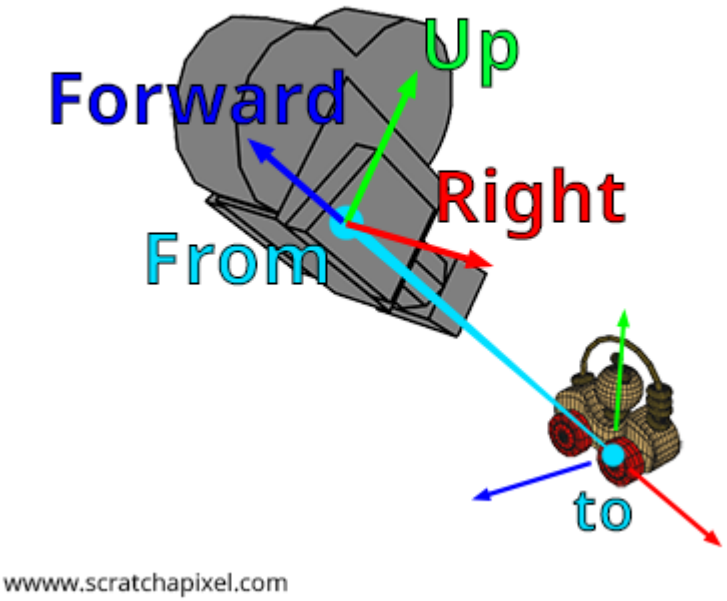


Figure 1: the local coordinate system of the camera aimed at a point.

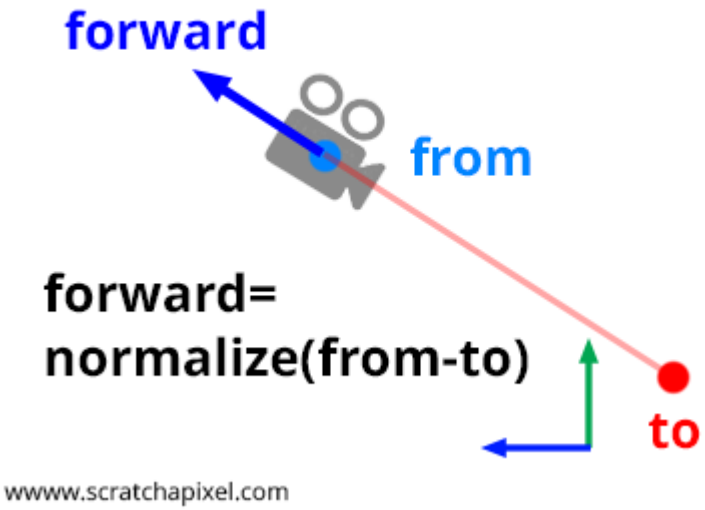


Figure 2: computing the forward vector from the position of the camera and target point.

Remember that a 4x4 matrix encodes the three axes of a Cartesian coordinate system. If this is not obvious to you, please read the lesson on Geometry. Remember that there are two conventions you need to pay attention to when you deal with matrices and coordinate systems. For matrices, you need to choose between row-major and column-major representations. Let's use the **row-major** notation. As for the coordinate system, you need to choose between the right-hand and the left-hand coordinate systems. Let's use a **right-hand** coordinate system. The fourth row of the 4x4 matrix (in a row-major matrix representation) encodes translation values.

RightxRightyRightz0UpxUpyUpz0ForwardxForwardyForwardz0TxTyTz1

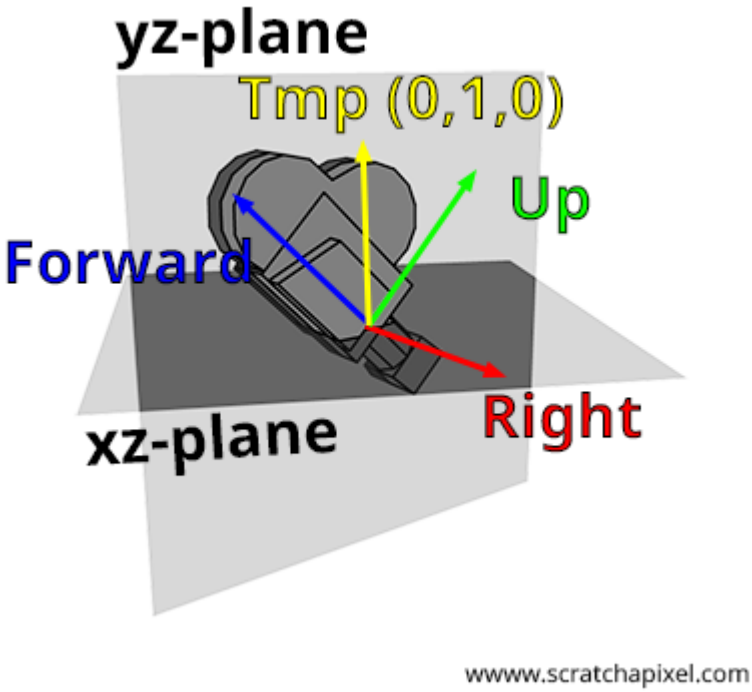


Figure 3: the vector (0,1,0) is in the plane defined by the forward and up vector. The vector perpendicular to this plane is thus the right vector.

How you name the axis of a Cartesian coordinate system is entirely up to you. You can call them x, y and z but in this lesson for clarity, we will name them **right** (for the x-axis), **up** (for the y-axis) and **forward** for the (z-axis). This is illustrated in figure 1. The method of building a 4x4 matrix from the from-to pair of points can be broken down into four steps:

- **Step 1: Compute the forward axis**. In Figures 1 and 2, it is quite easy to see that the forward axis of the camera's local coordinate system is aligned along the line segment defined by the points *from* and *to*. A little bit of geometry suffices to calculate this vector. You just need to normalize the vector From-To. Mind the direction of this vector: it is From-To not To-From). This can be done with the following code snippet:

  ```
  Vec3f forward = Normalize(From - to);
  ```

  Let's now calculate the over two vectors.

- **Step 2: Compute the right vector.** Recall from the lesson on Geometry that Cartesian coordinates are defined by three unit vectors that are perpendicular to each other. We also know that if we take two vectors A and B, they can be seen as lying in a plane. Furthermore, the cross product of these two vectors creates a third vector C perpendicular to that plane and thus perpendicular to both A and B. We can use this property to create the right vector. The idea here is to use some *arbitrary vector* and calculate the cross vector between the forward vector and this arbitrary vector. The result is a vector that is necessarily perpendicular to the forward vector and that can be used in the construction of our Cartesian coordinate system as the right vector. The code for computing this vector is simple since it only implies a cross-product between the forward vector and this arbitrary vector:

  ```
  Vec3f right = crossProduct(randomVec, forward);
  ```

  How do we choose this *arbitrary* vector? Well, this vector can't be arbitrary which is the reason why we wrote the word in italic. Think about this: if the forward vector is (0,0,1), then the right vector ought to be (1,0,0). This can only be done if we choose as our arbitrary vector, the vector (0,1,0). Indeed: (0,1,0) x (0,0,1) = (1,0,0) where the sign x here accounts for the cross product. Remember that the code/equation to compute the cross-product is:

  $$cx=ay*bz-az*by, cy=az*bx-ax*bz, cz=ax*by-ay*bx$$

  where a and b are two vectors and c is the result of the cross product of a and b. When you look at figure 3, you can also notice that regardless of the forward vector's direction, the vector perpendicular to the plane defined by the forward vector and the vector (0,1,0) is always the right vector of the camera's Cartesian coordinate system. That's great because the vector (0,1,0) can be used as our *arbitrary vector* (for now).

  ▼ Details

  Note also from that observation that the right vector always lies in the xz-plane. How come you may ask? If the camera has a roll wouldn't the right vector be in a different plane? That's true, but applying a roll to the camera is not something you can do directly with the look-at method. To add a camera roll, you would first need to create a matrix to roll the camera (rotate the camera around the z-axis) and then multiply this matrix by the camera-to-world matrix built with the look-at method.

  Finally, here is the code to compute the right vector:

  ```
  Vec3f tmp(0, 1, 0); Vec3f right = crossProduct(tmp, forward);
  ```

  Pay attention to the order of the vectors in the cross-product. Keep in mind that the cross-product is not commutative (it is anti-commutative, check the lesson on Geometry for more details). The best mnemonic way of remembering the right order is to think of the cross product of the forward vector (0,0,1) with the up vector (0,1,0) we know it should give (1,0,0) and not (-1,0,0). If you know the equations of the cross-product, you should easily find out that the order is up×forward and not the other way around. Great we have the forward and right vectors. Let's find the "true" up vector.

- **Step 3: Compute the up vector.** Well this is very simple, we have two orthogonal vectors, the forward and right vector, so computing the cross product between these two vectors will just give us the missing third vector, the up vector. Note that if the forward and right vector is normalized, then the resulting up vector computed from the cross product will be normalized too (The magnitude of the cross product of u and v is equal to the area of the parallelogram determined by u and v $\|u \times v\| = \|u\| \cdot \|v\| \cdot \sin\theta$):

  ```
  Vec3f up = crossProduct(forward, right);
  ```

  Here again, you need to be careful about the order of the vectors involved in the cross-product. Great, we now have the three vectors defining the camera coordinate system. Let's now build our final 4x4 camera-to-world matrix.

- **Step 4: set the 4x4 matrix using the right, up, and forward vector as from point.** All there is to do to complete the process is to build the camera-to-world matrix itself. For that, we just replace each row of the matrix with the right data:

  - Row 1: replace the first three coefficients of the row with the coordinates of the *right* vector,
  - Row 2: replace the first three coefficients of the row with the coordinates of the *up* vector,
  - Row 3: replace the first three coefficients of the row with the coordinates of the *forward* vector,
  - Row 4: replace the first three coefficients of the row with the coordinates of the *from* point.

Again, if you are unsure about why we do that, check the lesson on Geometry. Finally here is the source code of the complete function. It computes and returns a camera-to-world matrix from two arguments, the *from* and *to* points. Note that the function's third parameter (_up_) is the *arbitrary* vector used in the computation of the right vector. It is set to (0,1,0) in the main function but you may have to normalize it for safety (in case a user would input a non-normalized vector).

```
#include <cmath>
#include <cstdint>
#include <iostream>

struct float3
{
public:
    float x{ 0 }, y{ 0 } , z{ 0 };
    float3 operator - (const float3& v) const
    { return float3{ x - v.x, y - v.y, z - v.z }; }
};

void normalize(float3& v)
{
    float len = std::sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
    std::cout << v.x << " " << v.y << " " << v.z << "\n";
    v.x /= len, v.y /= len, v.z /= len;
    std::cout << v.x << " " << v.y << " " << v.z << "\n";
}

float3 cross(const float3& a, const float3& b)
{
    return {
        a.y * b.z - a.z * b.y,
        a.z * b.x - a.x * b.z,
        a.x * b.y - a.y * b.x
    };
}

struct mat4
{
public:
    float m[4][4] = {{1, 0, 0, 0},  {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}};
    float* operator [] (uint8_t i) { return m[i]; }
    const float* operator [] (uint8_t i) const { return m[i]; }
    friend std::ostream& operator << (std::ostream& os, const mat4& m)
    {
        return os << m[0][0] << ", " << m[0][1] << ", " << m[0][2] << ", " << m[0][3] << ", "
                  << m[1][0] << ", " << m[1][1] << ", " << m[1][2] << ", " << m[1][3] << ", "
                  << m[2][0] << ", " << m[2][1] << ", " << m[2][2] << ", " << m[2][3] << ", "
                  << m[3][0] << ", " << m[3][1] << ", " << m[3][2] << ", " << m[3][3];
    }

};

void lookat(const float3& from, const float3& to, const float3& up, mat4& m)
{
    float3 forward = from - to;
    normalize(forward);
    float3 right = cross(up, forward);
    normalize(right);
    float3 newup = cross(forward, right);

    m[0][0] = right.x,   m[0][1] = right.y,   m[0][2] = right.z;
    m[1][0] = newup.x,   m[1][1] = newup.y,   m[1][2] = newup.z;
```

```
    m[2][0] = forward.x, m[2][1] = forward.y, m[2][2] = forward.z;
    m[3][0] = from.x,    m[3][1] = from.y,    m[3][2] = from.z;
}


int main()
{
    mat4 m;

    float3 up{ 0, 1, 0 };
    float3 from{ 1, 1, 1 };
    float3 to{ 0, 0, 0 };

    lookat(from, to, up, m);

    std::cout << m << std::endl;

    return 0;
}
```

Should produce:

```
0.707107, 0, -0.707107, 0, -0.408248, 0.816497, -0.408248, 0, 0.57735, 0.57735, 0.57735, 0, 1, 1, 1, 1
```

The Look-At Method Limitations

The method is very simple and works generally well. Though it has a weakness. When the camera is vertical looking straight down or straight up, the forward axis gets very close to the arbitrary axis used to compute the right axis. The extreme case is of course when the forward axis and this arbitrary axis are perfectly parallel e.g. when the forward vector is either (0,1,0) or (0,-1,0). Unfortunately, in this particular case, the cross-product fails to produce a result for the right vector. There is no real solution to this problem. You can either detect this case and choose to set the vectors by hand (since you know what the configuration of the vectors should be anyway). A more elegant solution can be developed using quaternion interpolation.

Found a problem with this page?

Want to fix the problem yourself? Learn how to contribute!

Source this file on GitHub

Report a problem with this content on GitHub