GRAPHICS PROGRAMMING I

INTRODUCTION TO RAY TRACING

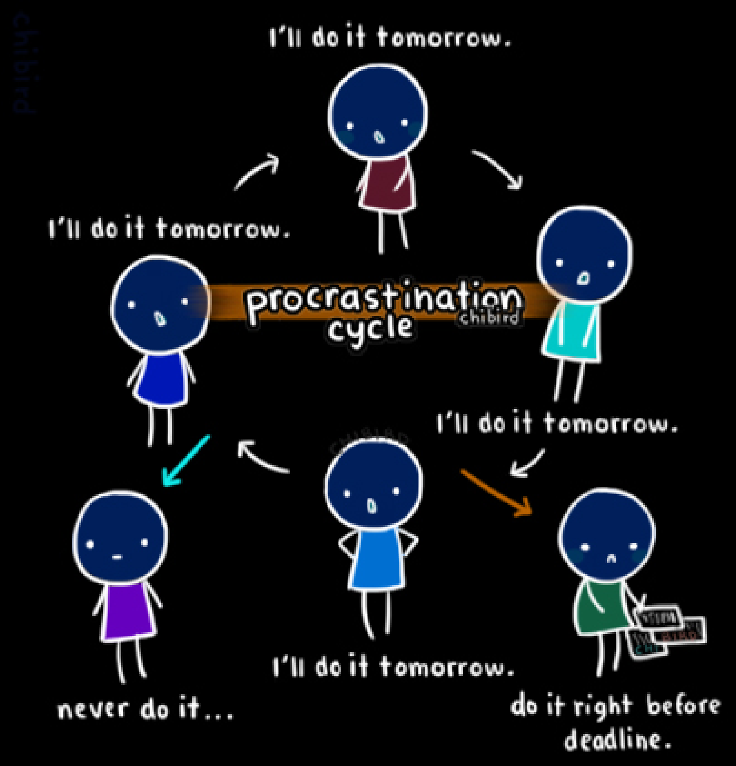
Welcome!

Thomas Goussaert
Thomas.goussaert@howest.be

Koen Samyn
koen.Samyn@howest.be

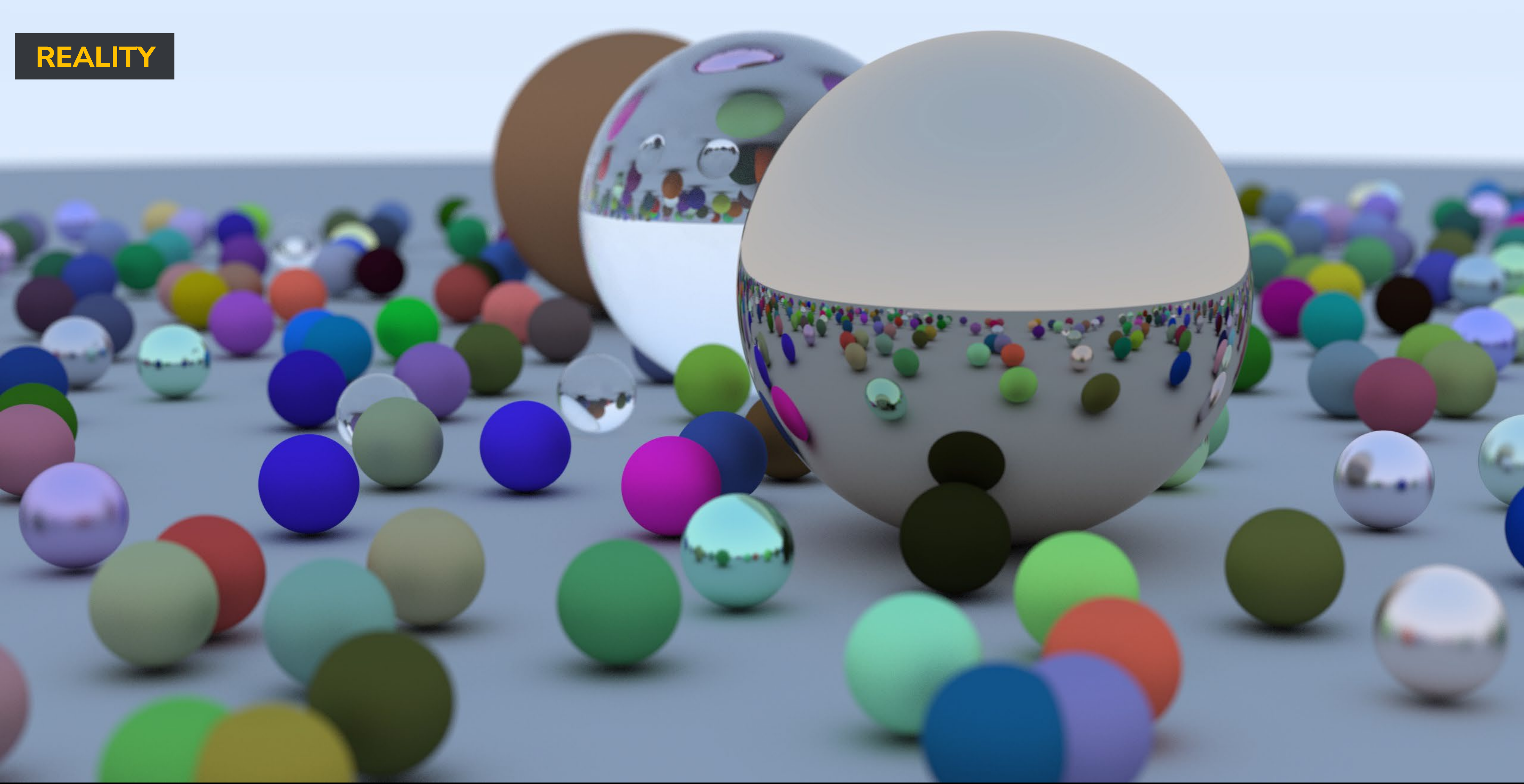
Welcome!

- Grading
  - Projects : 50% → There might be milestones for some projects!
    - Ray Tracer : 25%
    - Rasterizer: 25%
  - Exam : 50%
    - Theory : 25%
    - Project : 25% (more information later)
  - Retake? : 100%
    - Projects : 65%
    - Theory : 35%
- Tips:
  - Do not procrastinate! Work on your project every week!
  - Don't be afraid to refactor your code when necessary.
  - Version Control (mandatory) → Backup + Rollback.
  - Do <u>not</u> copy/paste code from others (incl. the internet)!



EXPECTATIONS

REALITY

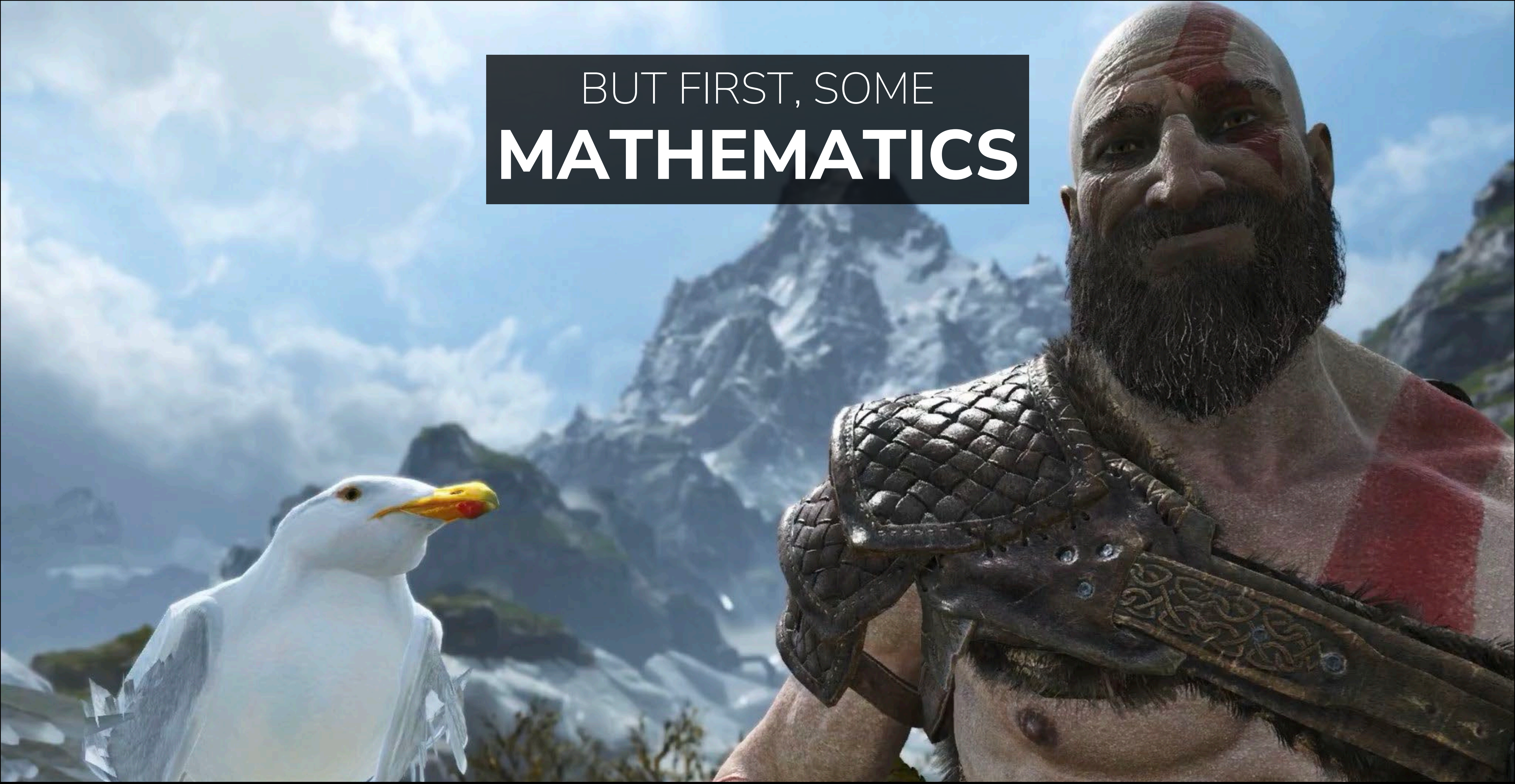


REALITY

What is Graphics Programming?

- Graphics Programming is all about:
  - solving the visibility problem
  - simulating the behavior of light
  - handling resources

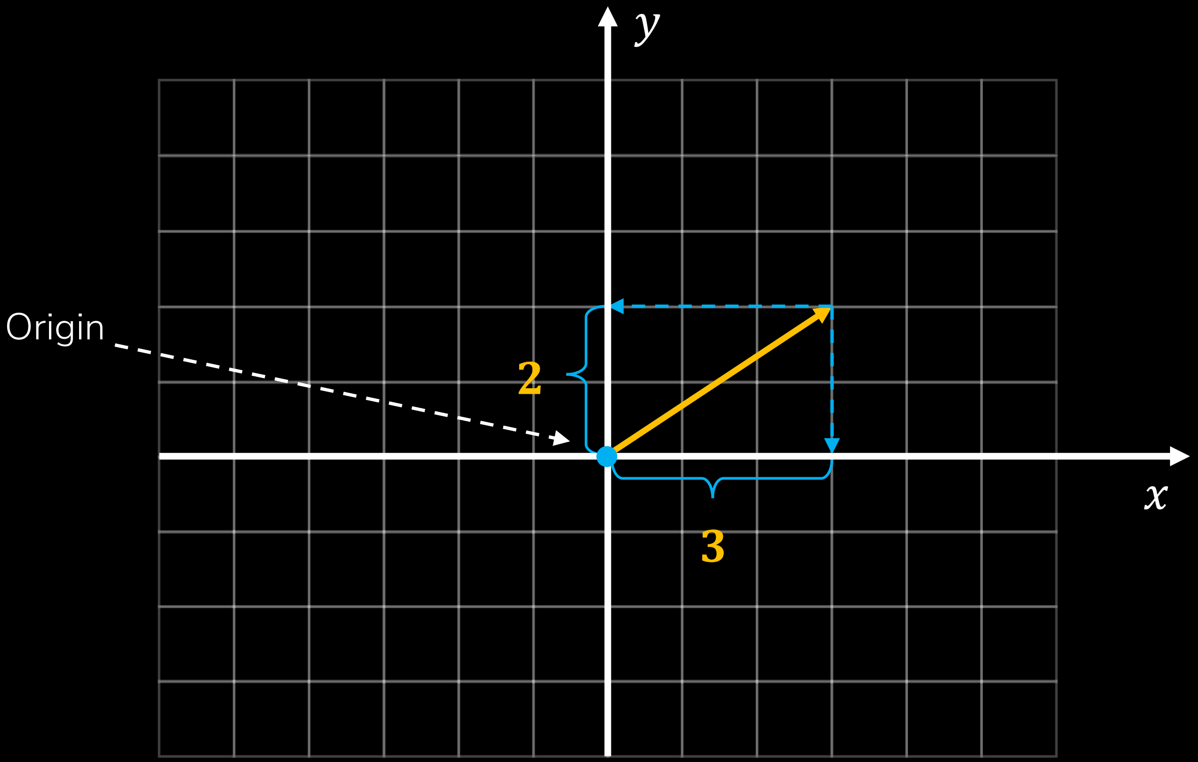
- There are two techniques we are going to discuss during this course:
  - Ray Tracing (Software)
  - Rasterization (Software & Hardware)



BUT FIRST, SOME **MATHEMATICS**

Mathematics: Linear Algebra - Vector

- What is a **vector**?
  - Physics → arrows pointing in space (length & direction)
  - Computer Science → ordered lists of numbers
  - Mathematics → generalizes both views…



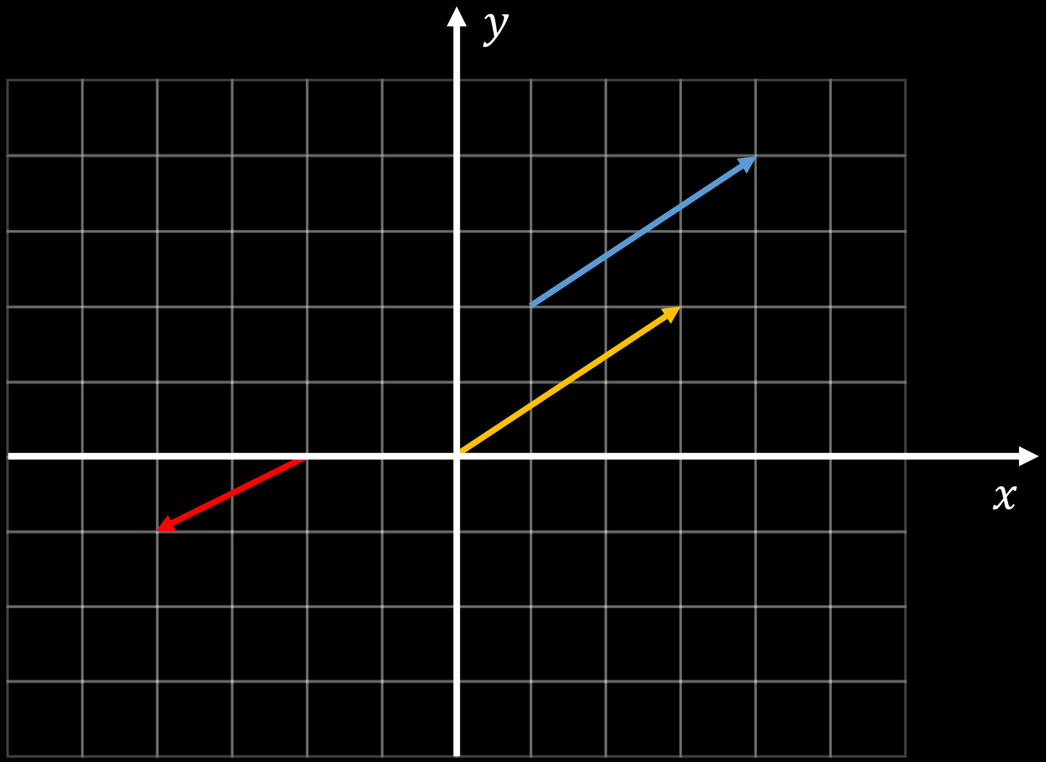
- Projecting a change in distance over a certain axis

$$\begin{bmatrix} 3 & * x \\ 2 & * y \end{bmatrix} \downarrow \begin{bmatrix} 3 & * 1 \\ 2 & * 1 \end{bmatrix} \downarrow \begin{bmatrix} 3 \\ 2 \end{bmatrix} \longrightarrow [3,2]$$

Column vs Row

Mathematics: Linear Algebra - Vector

- What is a vector?
  - Physics → arrows pointing in space (length & direction)
  - Computer Science → ordered lists of numbers
  - Mathematics → generalizes both views…



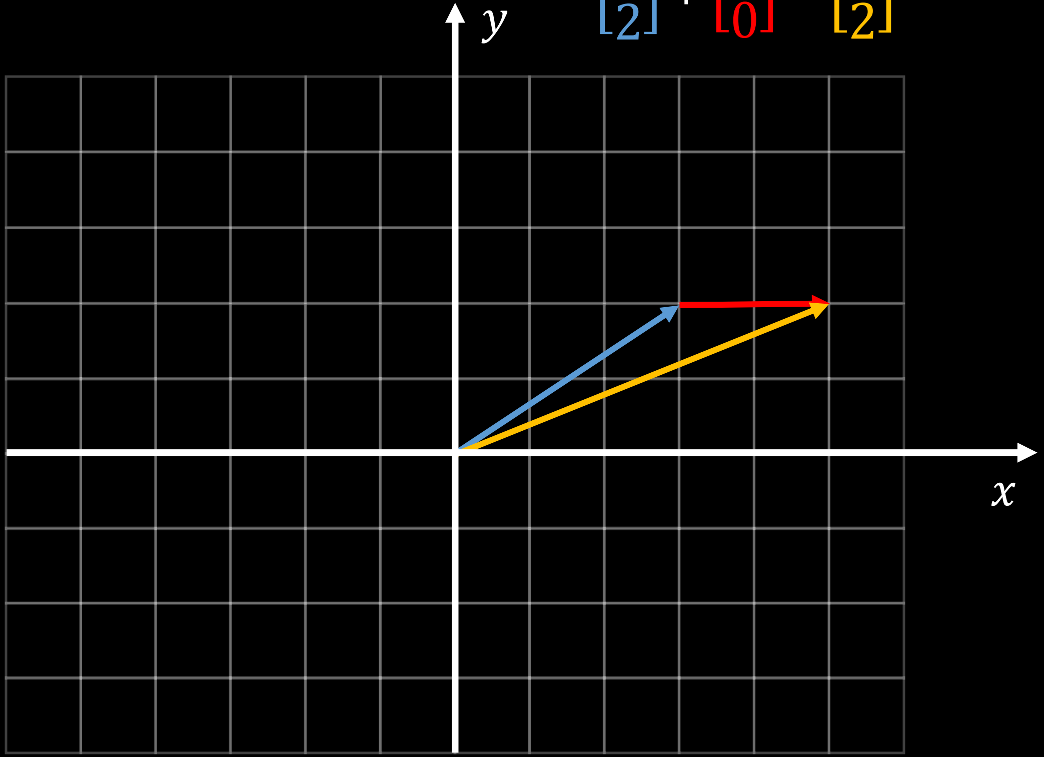
- There is no location. For positions we use Points

$$\begin{bmatrix}3\\2\end{bmatrix} == \begin{bmatrix}3\\2\end{bmatrix} \; != \begin{bmatrix}-2\\1\end{bmatrix}$$

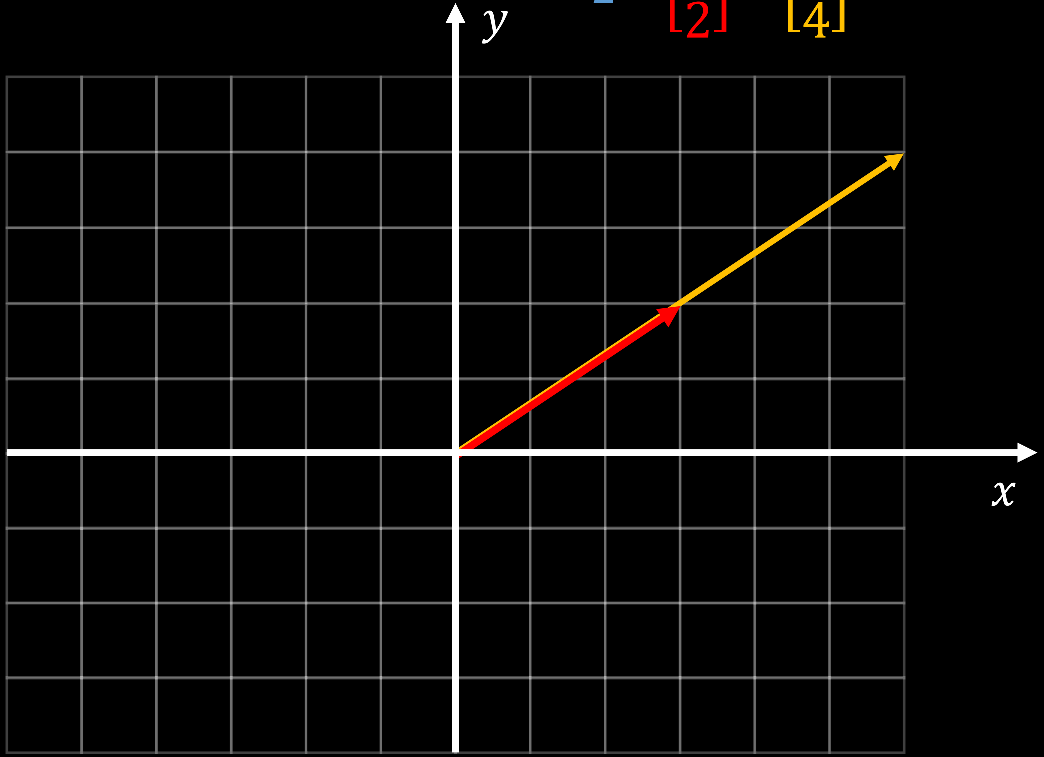
Mathematics: Linear Algebra - Vector

- Vector arithmetic

$$\begin{bmatrix}3\\2\end{bmatrix}+\begin{bmatrix}2\\0\end{bmatrix}=\begin{bmatrix}5\\2\end{bmatrix}$$

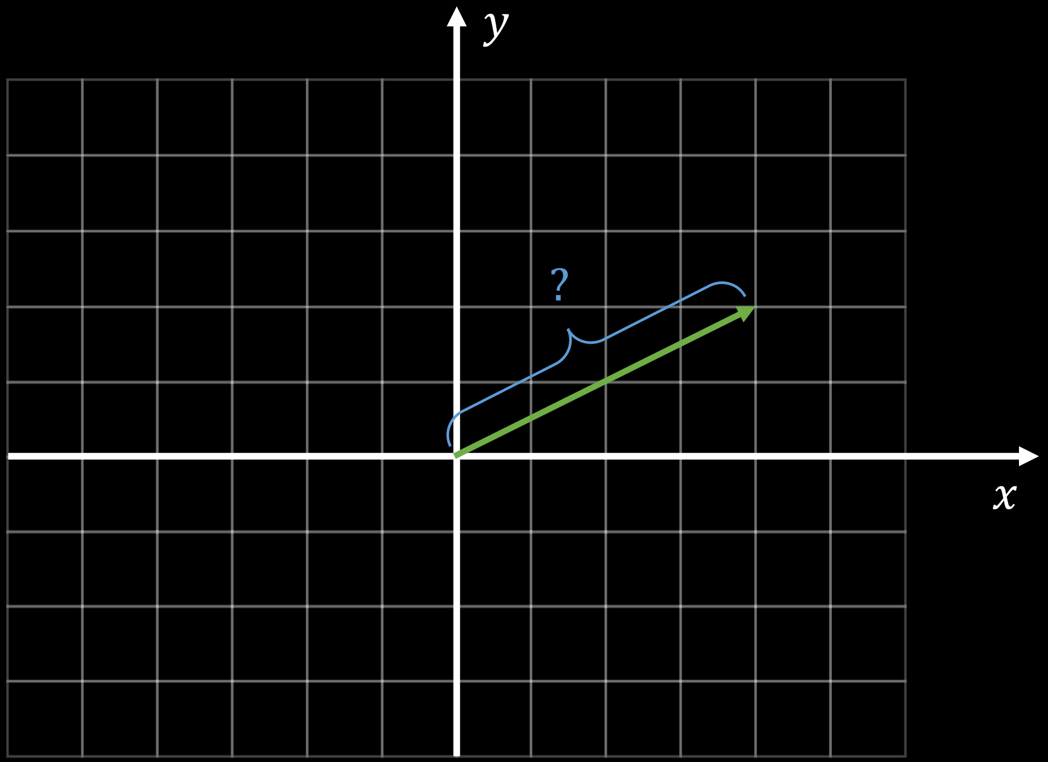


$$2*\begin{bmatrix}3\\2\end{bmatrix}=\begin{bmatrix}6\\4\end{bmatrix}$$



Mathematics: Linear Algebra - Vector

- So we know how to represent direction, what about length?
  - Most libraries have a function called Magnitude() or Length(), in Euclidean space!
  - This function can have performance implications! → Sqrt()



$$\sqrt{x^2 + y^2 + z^2}$$

$$\downarrow$$

$$\sqrt{4^2 + 2^2}$$

$$\downarrow$$

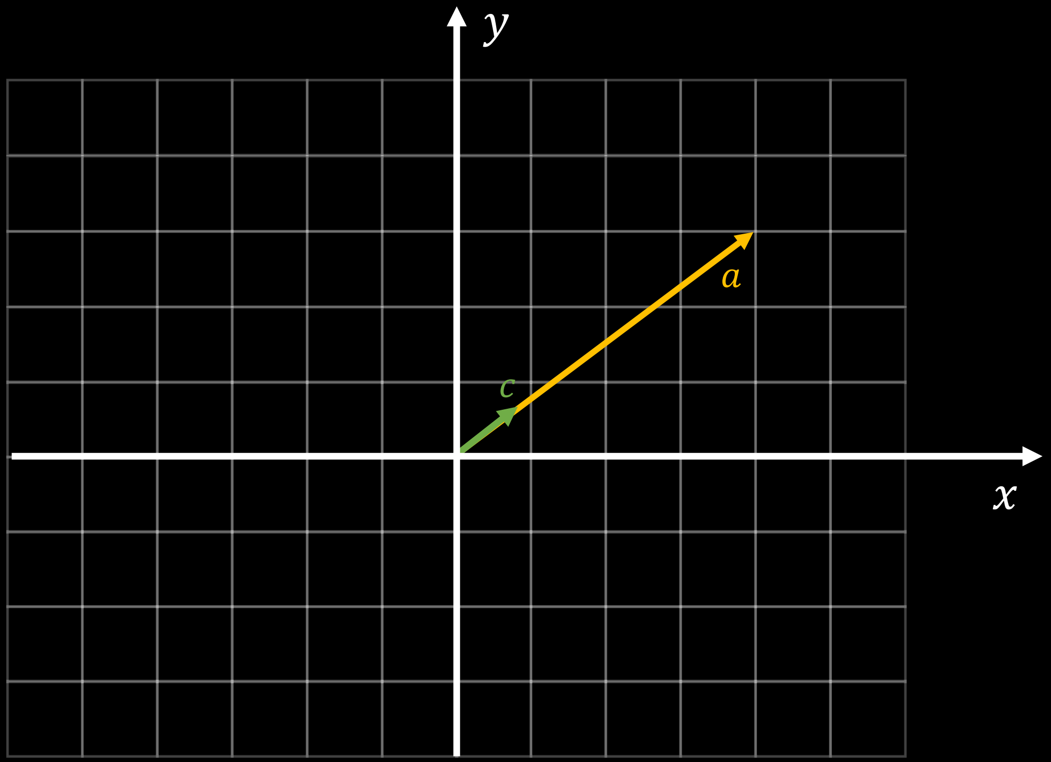
$$\sqrt{16 + 4}$$

$$\downarrow$$

$$\pm 4.47$$

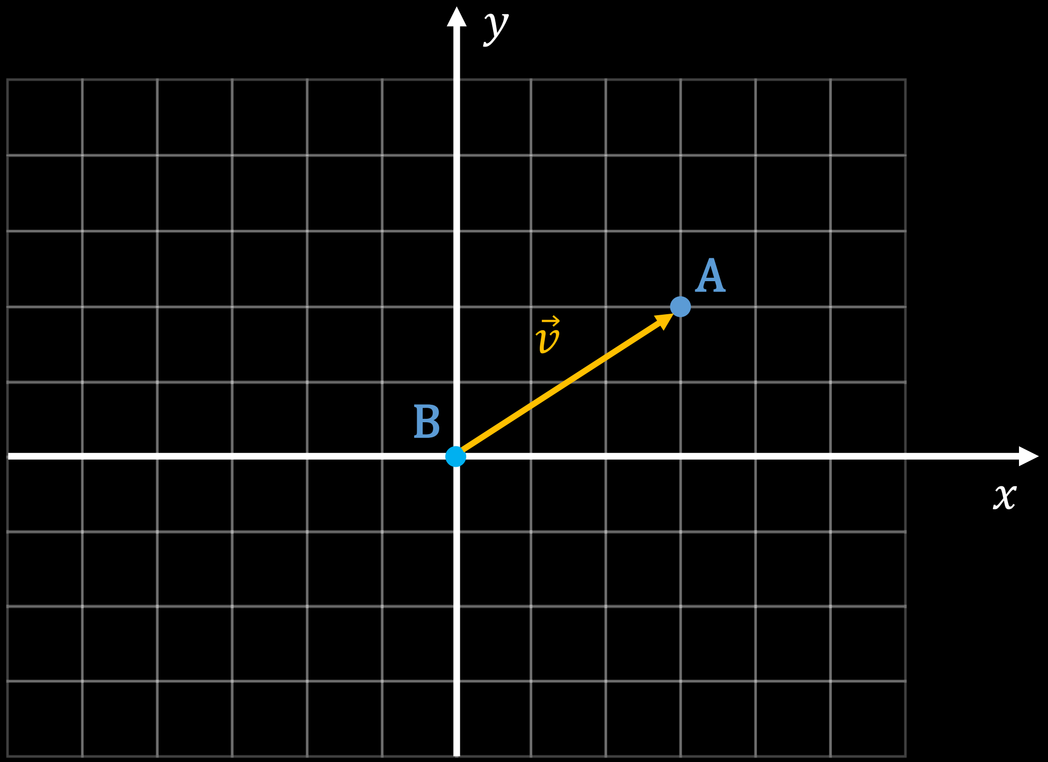
Mathematics: Linear Algebra - Vector

- Normalizing a vector: unit vector
  - result is a vector with length 1, pointing in the same direction: $\mathbf{c} = \frac{\boldsymbol{a}}{\|\boldsymbol{a}\|}$



Mathematics: Linear Algebra - Point

- So what about **points**?
  - We use them to represent locations in space.
  - We can use vector arithmetic on points, but not between points!



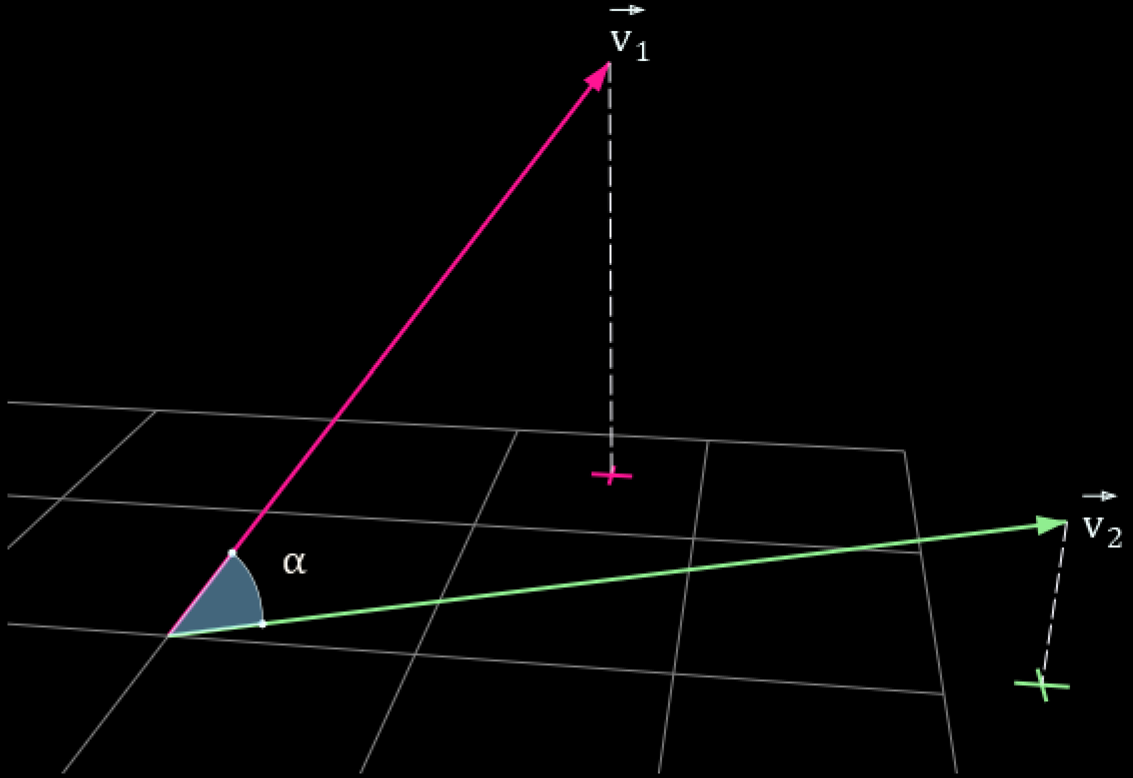
$$\mathrm{A} - \mathrm{B} = \vec{v}$$

$$\downarrow$$

$$\begin{bmatrix}3\\2\end{bmatrix} - \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}3\\2\end{bmatrix}$$

Mathematics: Linear Algebra - Vector

- There are some other very important features!
  - The dot product, or scalar product, is the product of the Euclidean magnitudes of two vectors and the cosine of the angle between them (geometrically).



The dot product is defined as :

$$\vec{v}_1 \cdot \vec{v}_2 = |\vec{v}_1||\vec{v}_2| \cos(\alpha)$$

AND also:

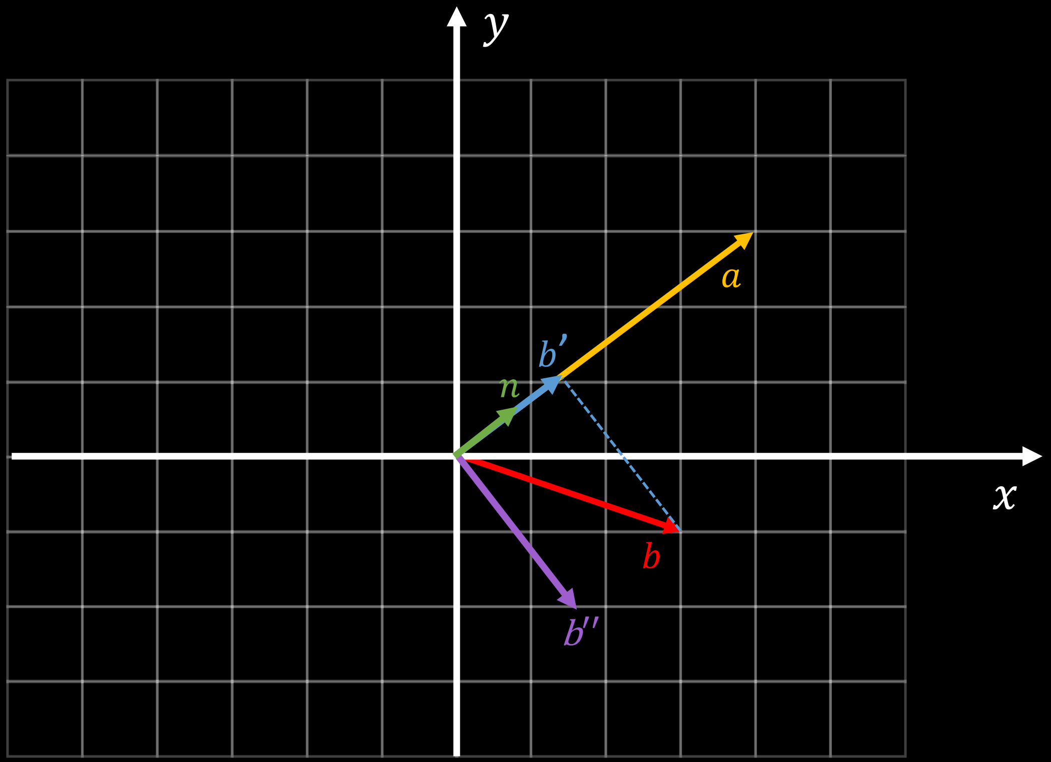
$$\vec{v}_1 \cdot \vec{v}_2 = v_{1x} v_{2x} + v_{1y} v_{2y} + v_{1z} v_{2z}$$

$$\downarrow$$

$$\alpha = \arccos\left(\frac{\overrightarrow{v_1} \cdot \overrightarrow{v_2}}{\|\overrightarrow{v_1}\| \|\overrightarrow{v_2}\|}\right)$$

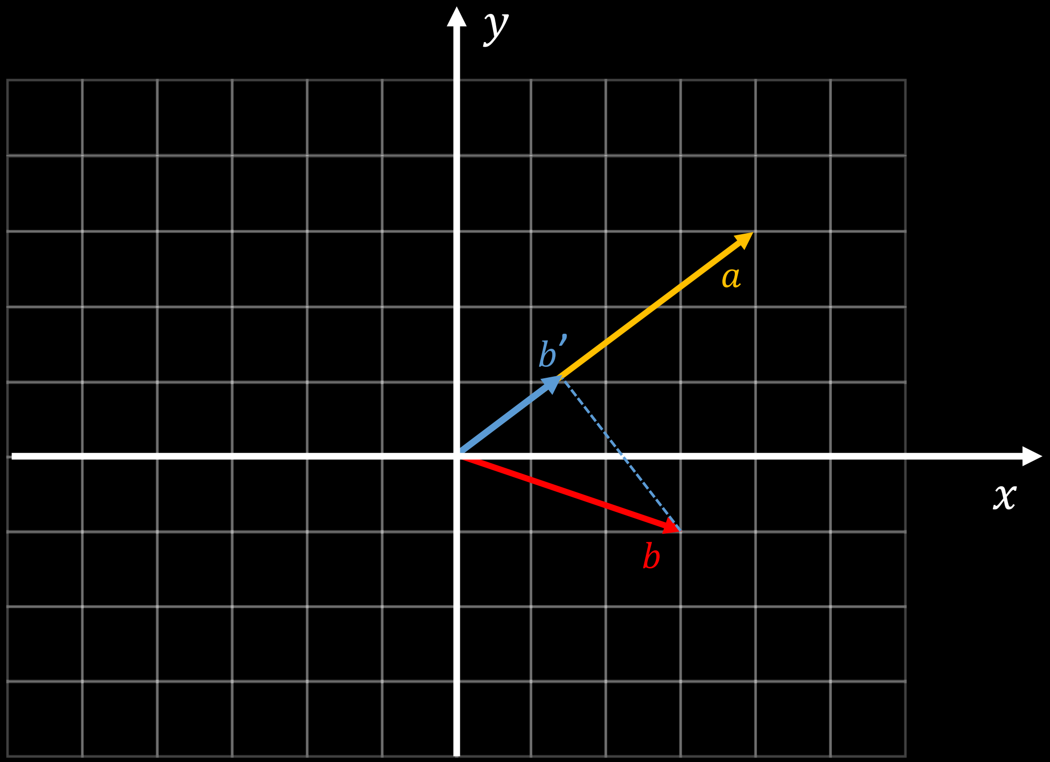
Mathematics: Linear Algebra - Vector

- Decomposition of vector b onto vector a
  - n is the normalized vector a: $\boldsymbol{n} = \frac{\boldsymbol{a}}{\|\boldsymbol{a}\|}$
  - vector projection of vector b onto vector a: $\boldsymbol{b'} = (\boldsymbol{n} \,.\, \boldsymbol{b})\, \boldsymbol{n}$
  - vector rejection of vector b onto vector a: $\boldsymbol{b''} = \boldsymbol{b} - \boldsymbol{b'}$



Mathematics: Linear Algebra - Vector

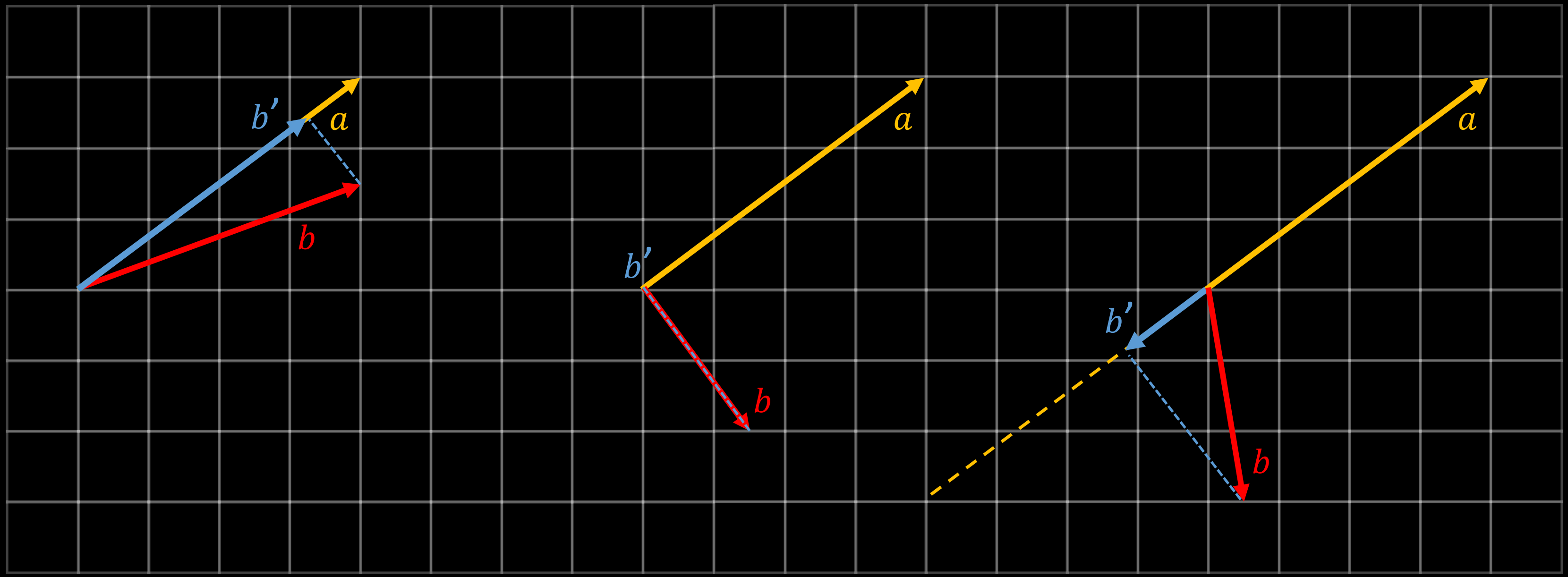
- There are some other very important features!
  - The dot product can also be seen as: $\|\mathbf{project}(\boldsymbol{b})\|\ \|\boldsymbol{a}\|$ or $\|\mathbf{b}'\|\ \|\boldsymbol{a}\|$
  - What is the equivalent of doing Dot(a, a)? ☺



Mathematics: Linear Algebra - Vector

- There are some other very important features!
  - The dot product can also be seen as: $\|\mathbf{project}(\boldsymbol{b})\|\ \|\boldsymbol{a}\|$ or $\|\mathbf{b}'\|\ \|\boldsymbol{a}\|$
  - This means the range is not necessarily [-1,1] → This is only true if both vectors are normalized!

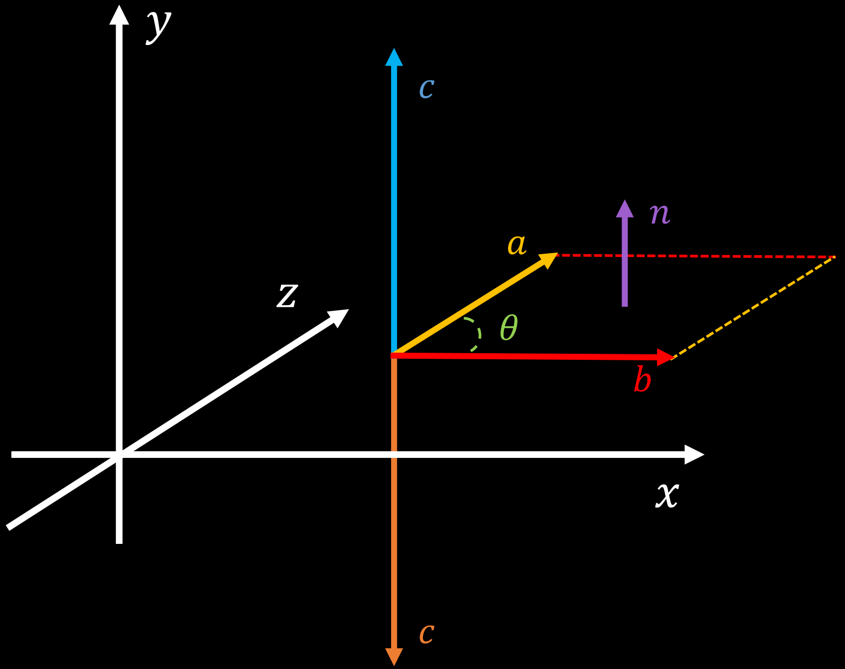
$\|\boldsymbol{b}'\|\,.\,\|\boldsymbol{a}\| > \mathbf{0}$ $\|\boldsymbol{b}'\|\,.\,\|\boldsymbol{a}\| = \mathbf{0}$ $\|\boldsymbol{b}'\|\,.\,\|\boldsymbol{a}\| < \mathbf{0}$



Mathematics: Linear Algebra - Vector

- There are some other very important features!
  - The cross product, or vector product, is the product of the Euclidean magnitudes of two vectors, the sine of the angle between them and the unit vector perpendicular to the plane containing the two vectors (geometrically).

$$\boldsymbol{a} \times \boldsymbol{b} = \{\boldsymbol{a.y}\,\boldsymbol{b.z} - \boldsymbol{a.z}\,\boldsymbol{b.y}, \boldsymbol{a.z}\,\boldsymbol{b.x} - \boldsymbol{a.x}\,\boldsymbol{b.z}, \boldsymbol{a.x}\,\boldsymbol{b.y} - \boldsymbol{a.y}\,\boldsymbol{b.x}\}$$



$$\boldsymbol{a} \times \boldsymbol{b} = \|\boldsymbol{a}\|\|\boldsymbol{b}\|\,\boldsymbol{sin}\,\boldsymbol{\theta}\,\boldsymbol{n}$$

$$\downarrow$$

Left-Handed Coordinate System

$$c = a \times b$$
$$c = b \times a$$

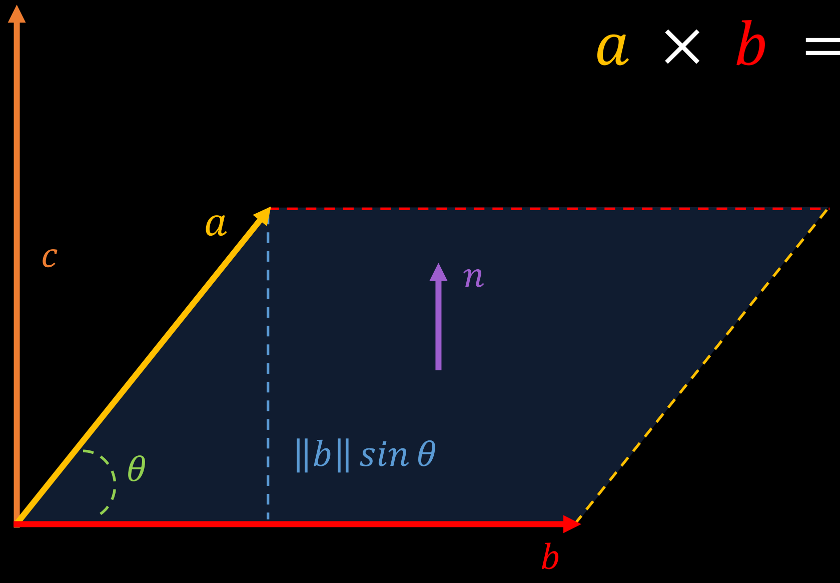
Right-Handed Coordinate System

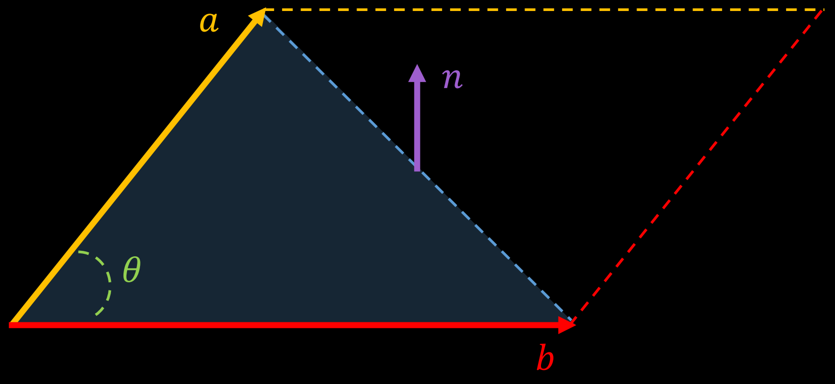
$$c = a \times b$$
$$c = b \times a$$

Mathematics: Linear Algebra - Vector

- There are some other very important features!
  - The cross product, or vector product, is the product of the Euclidean magnitudes of two vectors, the sine of the angle between them and the unit vector perpendicular to the plane containing the two vectors (geometrically).

$$a \times b = \|\boldsymbol{a}\|\|\boldsymbol{b}\| \, \boldsymbol{\sin\theta} \, \boldsymbol{n}$$





$$A = \|\boldsymbol{a}\|\|\boldsymbol{b}\| \, \boldsymbol{\sin\theta} \longrightarrow A = \|\boldsymbol{a} \times \boldsymbol{b}\|$$

$$A = \|\boldsymbol{c}\|$$

$$A\,\textbf{triangle} = \frac{1}{2}\|\boldsymbol{a} \times \boldsymbol{b}\|$$

Mathematics: Linear Algebra - Vector & Point

- So to be complete, these are the typical operators and functions between types and their results:
  - $u + v$ = vector
  - $u - v$ = vector
  - $a * v$ = vector
  - u/a = vector
  - $\mathrm{dot}(u, v)$ = scalar
  - $\mathrm{cross}(u, v)$ = vector
  - $\mathrm{magnitude}(u)$ = scalar
  - $\mathrm{normalize}(u)$ = vector
  - $\mathrm{project}(u, v)$ = vector
  - $\mathrm{reject}(u, v)$ = vector
  - $p + v$ = point
  - $p - v$ = point
  - $\mathrm{p} - \mathrm{p}$ = vector
  - $\mathrm{distance}(p, p)$ = scalar

Mathematics: Linear Algebra - Matrix

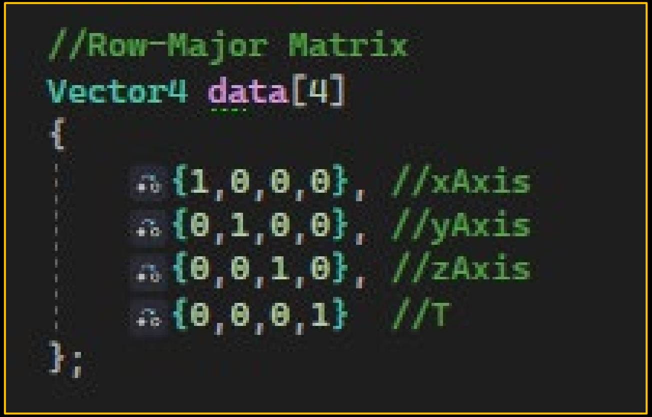
- A **matrix** is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns. [1]
  - Matrices can have different sizes and are often defined as matrix[n][m], where n = rows and m = columns.
  - When n == m, we talk about a square matrix.
  - A vector is also a matrix! It <u>can</u> be defined as matrix[1][m]. → row vector
  - Instead of defining a matrix as matrix[n][m], we can also define a matrix as matrix[m][n]! This is very important when we work with matrices because it influences the <u>order of evaluation</u>! We talk about:
    - Row-Major Order = matrix[n][m]
    - Column-Major Order = matrix[m][n]
  - We can switch between both forms by using the Transpose function. We denoted the transpose of matrix $\boldsymbol{M}$ as $\boldsymbol{M}^T$.

$$\begin{array}{c} \downarrow \quad\quad\quad\quad \\ \begin{bmatrix} \mathbf{0} & 3 & 6 \\ \mathbf{1} & 4 & 7 \\ \mathbf{2} & 5 & 8 \end{bmatrix} \end{array} \longrightarrow \begin{array}{cc} \rightarrow & \begin{bmatrix} \mathbf{0} & \mathbf{1} & \mathbf{2} \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \\ & \end{array}$$

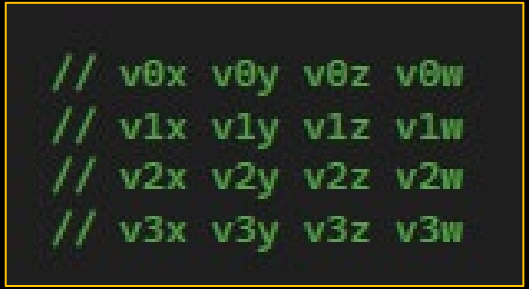
1 Foundations of Game Engine Development – Eric Lengyel

Mathematics: Linear Algebra - Matrix

- A **matrix** is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - So which form is correct? → Both!
  - We choose to use **Row-Major Matrices**.
  - How we interpret a matrix (column vs row) has **nothing** to do with the **memory layout**! To make things more interesting, storing column matrices in C++ can be bad for the **locality of the data**. Because of this we store the columns in the "rows" and vice versa. ☺



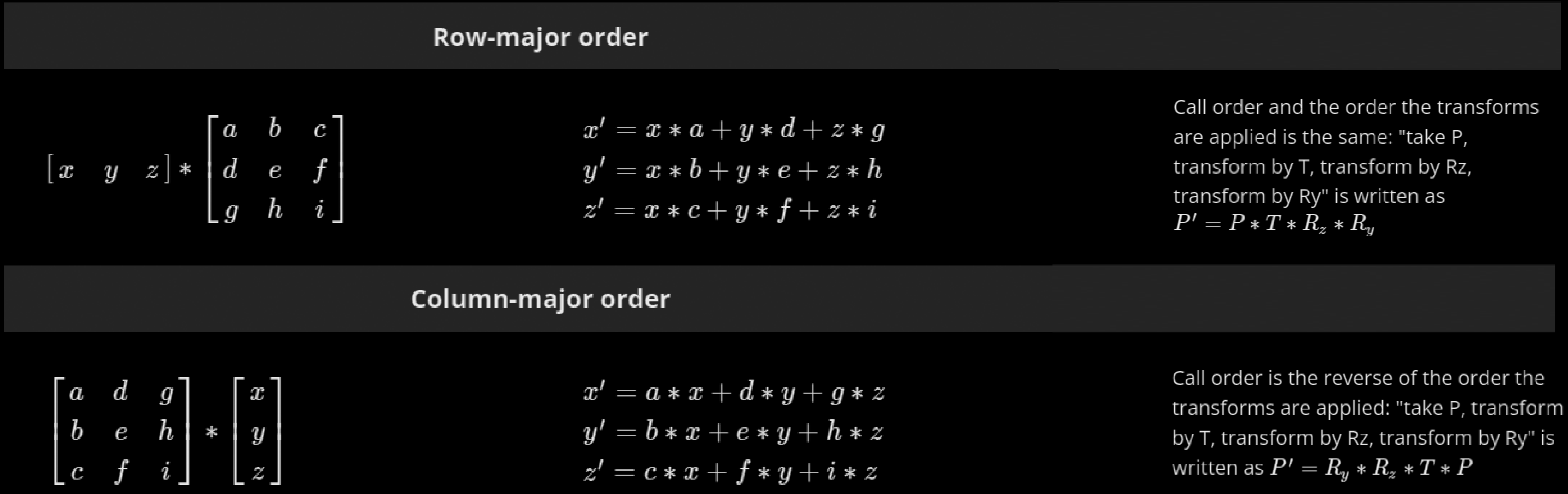
```
//Row-Major Matrix
Vector4 data[4]
{
        {1,0,0,0}, //xAxis
        {0,1,0,0}, //yAxis
        {0,0,1,0}, //zAxis
        {0,0,0,1}  //T
};
```



```
// v0x v0y v0z v0w
// v1x v1y v1z v1w
// v2x v2y v2z v2w
// v3x v3y v3z v3w
```

Mathematics: Linear Algebra - Matrix

- A matrix is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.



Row-major order		
$\begin{bmatrix} x & y & z \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$	$x' = x*a + y*d + z*g$ $y' = x*b + y*e + z*h$ $z' = x*c + y*f + z*i$	Call order and the order the transforms are applied is the same: "take P, transform by T, transform by Rz, transform by Ry" is written as $P' = P*T*R_z*R_y$
Column-major order		
$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$x' = a*x + d*y + g*z$ $y' = b*x + e*y + h*z$ $z' = c*x + f*y + i*z$	Call order is the reverse of the order the transforms are applied: "take P, transform by T, transform by Rz, transform by Ry" is written as $P' = R_y*R_z*T*P$

Mathematics: Linear Algebra - Matrix

- A **matrix** is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - The **identity matrix** is the matrix[n][n] (square matrix) whose entries on the main diagonal are all ones and whose entries everywhere else are all zeros. [1]
  - When multiplying a matrix $\boldsymbol{M}$ with an identity matrix $\boldsymbol{I}$, you always get back the original matrix $\boldsymbol{M}$. In other words, the identity matrix is a **default matrix** that has no effect. (often used for initialization)

$$\begin{array}{ccc}\left[\begin{array}{ccc}\mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1}\end{array}\right] & \begin{bmatrix}2 & 1 & 4 \\ -1 & 0 & 3 \\ 5 & 6 & 5\end{bmatrix} = & \begin{bmatrix}2 & 1 & 4 \\ -1 & 0 & 3 \\ 5 & 6 & 5\end{bmatrix}\end{array}$$

$$\begin{array}{ccc}\uparrow & \uparrow & \uparrow \\ \boldsymbol{x} & \boldsymbol{y} & \boldsymbol{z}\end{array} \longrightarrow$$

In other words, we use a matrix to define a **linear transformation**.

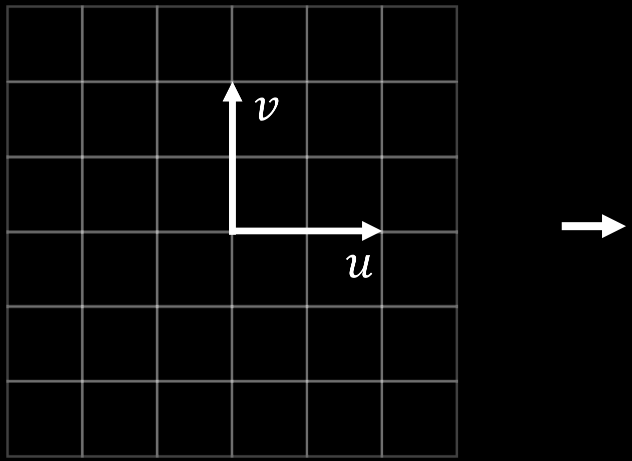
1 Foundations of Game Engine Development – Eric Lengyel

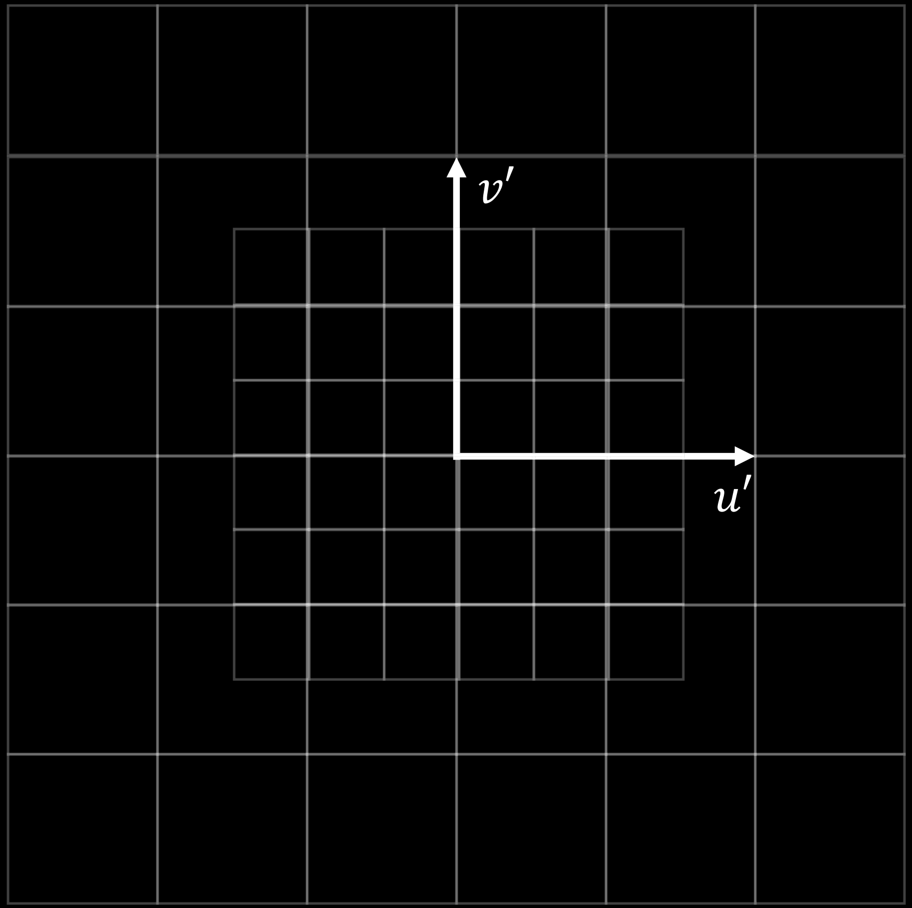
Mathematics: Linear Algebra - Matrix

$$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix}\mathrm{u}_x & u_y\\v_x & v_y\end{bmatrix}$$





$$\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix}\mathbf{2} * u_x & 0 * u_y\\0 * v_x & \mathbf{2} * v_y\end{bmatrix}$$

Mathematics: Linear Algebra - Matrix

- A **matrix** is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - What we just saw was a **linear transformation** where we just scale the base vectors. Using this matrix we can now transform all the points so they match the transformation.
    - Rule for linear transformation: **"grid lines" remain parallel and evenly spaced**! [1] Else it is not linear!

$$\begin{bmatrix} \mathbf{2} * \boldsymbol{u_x} & 0 * u_y \\ 0 * v_x & \mathbf{2} * \boldsymbol{v_y} \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Where $\boldsymbol{v'}$ "lands" after the transformation

Where $\boldsymbol{u'}$ "lands" after the transformation

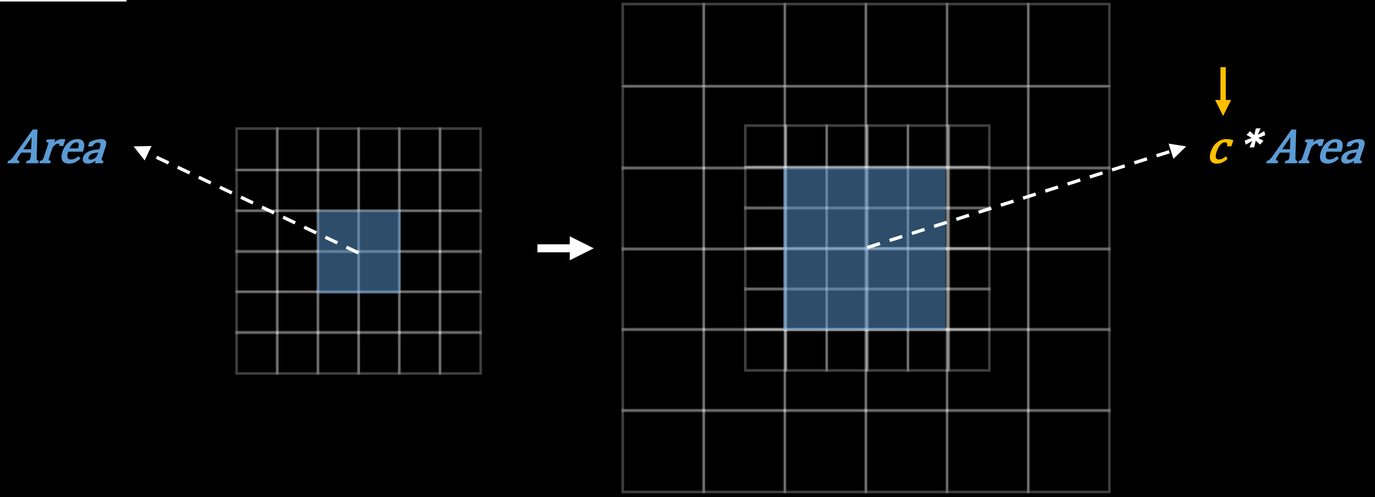
  - We see that there is a strong connection between the defined **origin**, the base vectors (**axis**) and the numerical values inside a matrix. We can also spot that the form of the matrix (row-major vs column-major) is important!

1 3Blue1Brown Series – Linear Algebra – Grant Sanderson

Mathematics: Linear Algebra - Matrix

- A matrix is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - When performing a linear transformation, we can do the opposite transformation by using the inverse of that matrix. Inverses are defined only for square matrices. We denoted the inverse of matrix $\boldsymbol{M}$ as $\boldsymbol{M}^{-1}$.
  - An inverse does not always exist! It only exists when the determinant of the matrix is not 0.

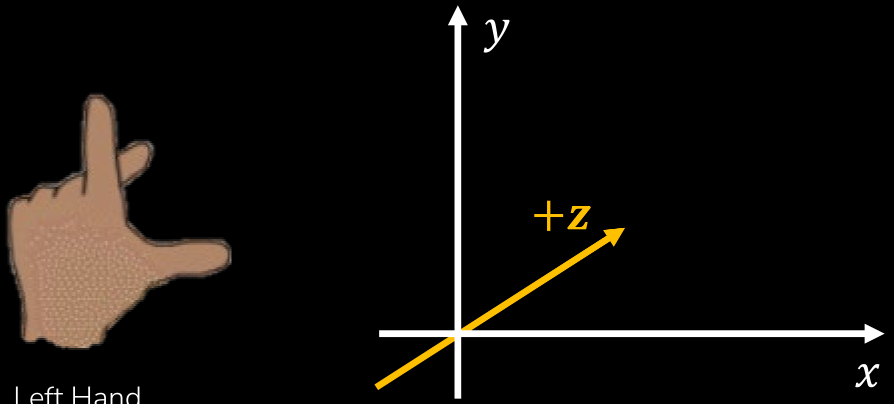
  - The determinant of a matrix[n][n] is a scalar value that can be thought of as a sort of magnitude for M. So it gives us the factor of how much an area increases or decreases due to the transformation. What does a factor of 0 mean?



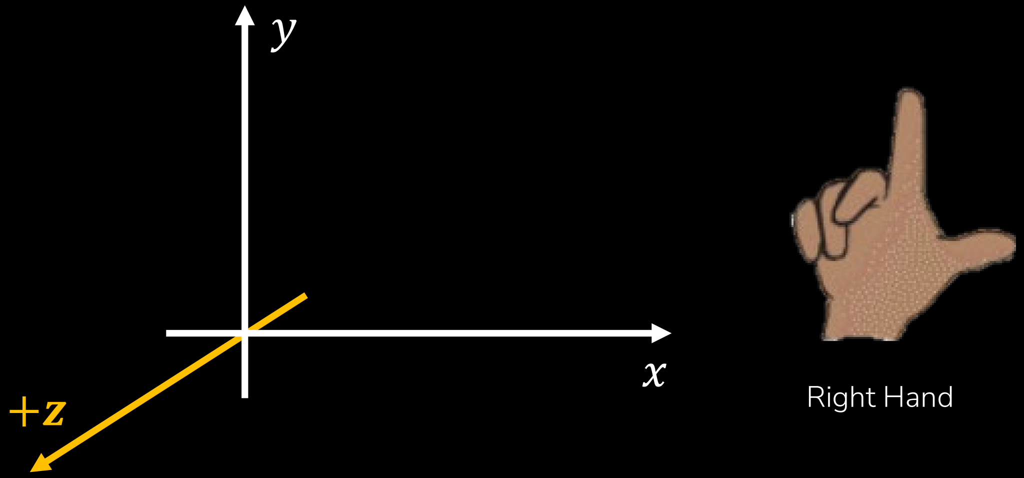
1 Foundations of Game Engine Development – Eric Lengyel

Mathematics: Linear Algebra - Matrix

- A matrix is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - We use matrices to store linear transformations. We've seen a transformation called scaling, but there are two other transformations: translation and rotation.
  - Before we talk about how we can construct these transformations, we have to briefly talk about our coordinate system (see cross product). The coordinate system determines how we are going to interpret some of our values. You have to pick one system and stick to it! We are going to use a Left-Handed Coordinate System! (Y-up)



Left-Handed Coordinate System



Right-Handed Coordinate System

Mathematics: Linear Algebra - Matrix

- A matrix is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - Rotations often occur in a local coordinate system in which the axis of rotation is aligned to one of the coordinate axes, but they can also be applied about an arbitrary axis specified by a direction vector.[1]
  - To follow common convention, we consider a rotation through a positive angle about an axis to be one that is a clockwise rotation when the axis points away from the viewer (aka left-handed coordinate system).[1]

$$M_{rot\ \boldsymbol{x}}(\theta) = \begin{bmatrix} \mathbf{1} & 0 & 0 \\ 0 & \boldsymbol{\cos\theta} & \boldsymbol{-\sin\theta} \\ 0 & \boldsymbol{\sin\theta} & \boldsymbol{\cos\theta} \end{bmatrix} \quad M_{rot\ \boldsymbol{y}}(\theta) = \begin{bmatrix} \boldsymbol{\cos\theta} & 0 & \boldsymbol{-\sin\theta} \\ 0 & \mathbf{1} & 0 \\ \boldsymbol{\sin\theta} & 0 & \boldsymbol{\cos\theta} \end{bmatrix} \quad M_{rot\ \boldsymbol{z}}(\theta) = \begin{bmatrix} \boldsymbol{\cos\theta} & \boldsymbol{\sin\theta} & 0 \\ \boldsymbol{-\sin\theta} & \boldsymbol{\cos\theta} & 0 \\ 0 & 0 & \mathbf{1} \end{bmatrix}$$

1 Foundations of Game Engine Development – Eric Lengyel

Mathematics: Linear Algebra - Matrix

- A matrix is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - Scaling can be represented as follows:

$$M_{scale}\,(\mathbf{s_x}, \mathbf{s_y}, \mathbf{s_z}) = \begin{bmatrix} \mathbf{s_x} & 0 & 0 \\ 0 & \boldsymbol{s_y} & 0 \\ 0 & 0 & \boldsymbol{s_z} \end{bmatrix}$$

  - What about translation? We use homogeneous coordinates (4D projective space) and incorporate the translations into a <u>matrix[4][4]</u>. [1]

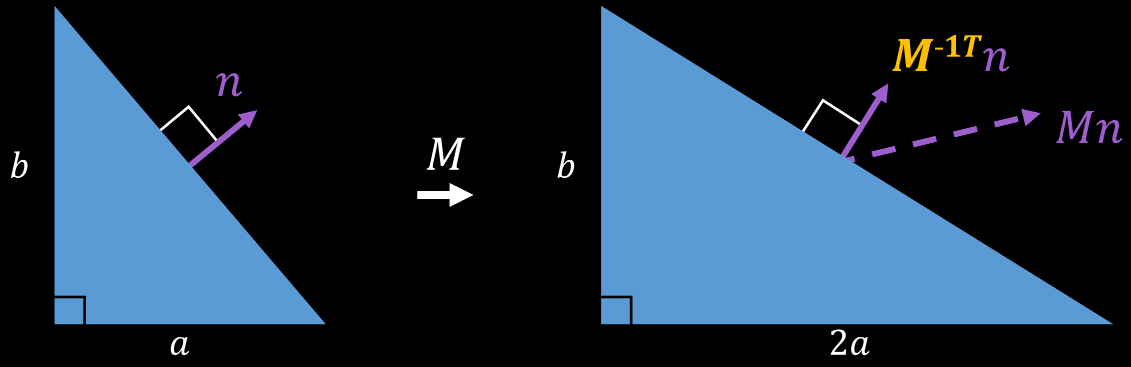
$$\begin{bmatrix} M & 0 \\ \boldsymbol{t} & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} M_{00} & M_{01} & M_{02} & 0 \\ M_{10} & M_{11} & M_{12} & 0 \\ M_{20} & M_{21} & M_{22} & 0 \\ \boldsymbol{t_x} & \boldsymbol{t_y} & \boldsymbol{t_z} & 1 \end{bmatrix}$$

1 Foundations of Game Engine Development – Eric Lengyel

Mathematics: Linear Algebra - Matrix

- A matrix is a mathematical object composed of a set of numerical quantities arranged in a two-dimensional array of rows and columns.
  - Because we'll be using an 4D matrix for transformations, we'll also need to use a 4D vector!
  - We make a distinction between transforming vectors, points and normal because they behave differently!

  - Vectors have <u>no location</u>, so translation is not necessary. To cancel out the translation component of a 4D matrix we can set the $\boldsymbol{w}$ component of the 4D vector to 0.
  - Points on the other hand do have a location. To take into account the translation we set the $\boldsymbol{w}$ component to 1.
  - What about normals? Are they not regular vectors?



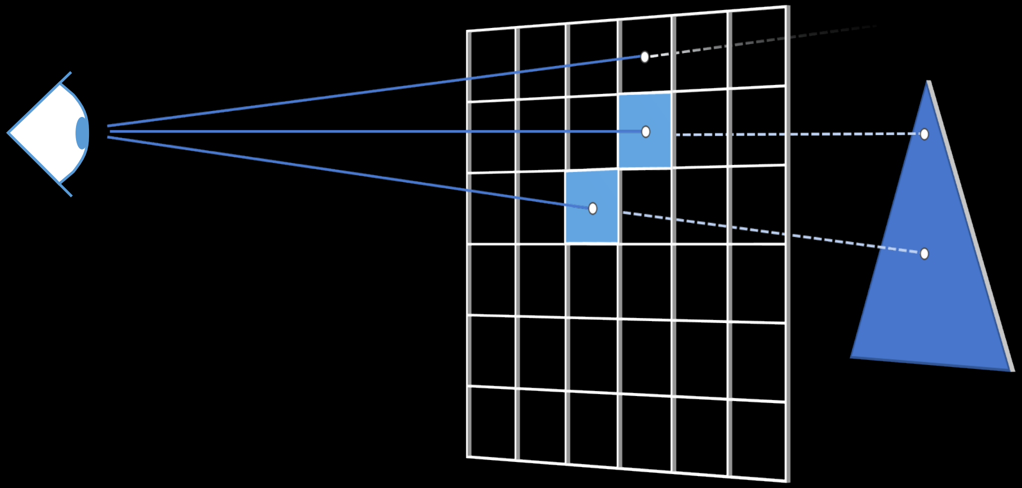
1 Foundations of Game Engine Development – Eric Lengyel

STILL WITH ME?

Ray Tracing: What is Ray Tracing?

Ray Tracing

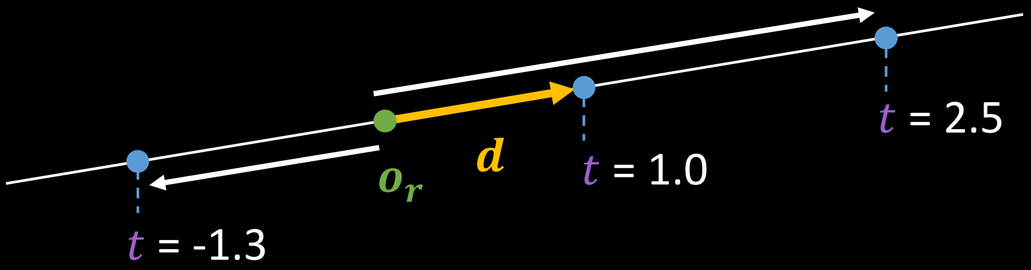
```
for each pixel (cast a ray)
    for each primitive (Triangle, Plane, Sphere, etc.)
        does ray hit primitive?
```



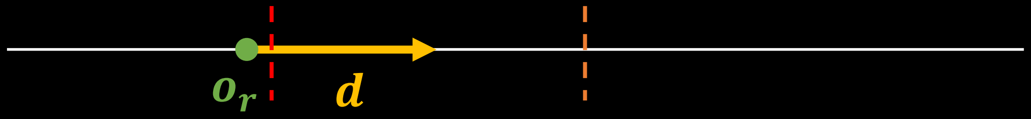
Ray Tracing: What is a Ray?

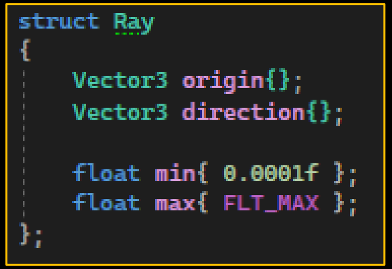
- A ray is a straight line that's defined by a point (origin) and a direction. It's often an infinite line, but we generally describe it with a half-open interval (eg. [0,∞)). We also refer to a ray as a parametrized line, because it can be written as:

$$\boldsymbol{p}(t) = \boldsymbol{o}_{\boldsymbol{ray}} + t\boldsymbol{d}$$



- The interval is usually defined with parameters tMin and tMax.





```
struct Ray
{
    Vector3 origin{};
    Vector3 direction{};

    float min{ 0.0001f };
    float max{ FLT_MAX };
};
```

Ray Tracing: Intersections – Ray-Plane?

- We know how we represent rays, but what about primitives or **surfaces**? We usually define them implicitly using **implicit equations**. This means we define a set of points on the surface as:

$$\boldsymbol{f(x, y, z) = 0}$$

- Any point (x, y, z) that is **on the surface returns zero** when given as arguments to the function f. Any point that is not on the surface returns a number other than zero.
- When we want to know where a ray and implicit surface intersect, we can plug in the implicit equation of a ray as follows:

$$\boldsymbol{f\big(p(t)\big) = 0}$$

$$\downarrow$$

$$\boldsymbol{f\big(o_{ray} + td\big) = 0}$$

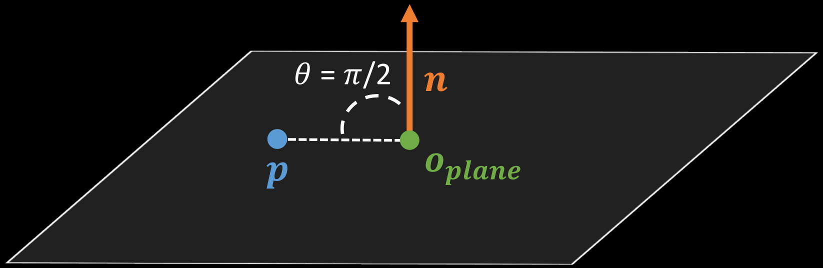
- Now let's consider an infinite plane…

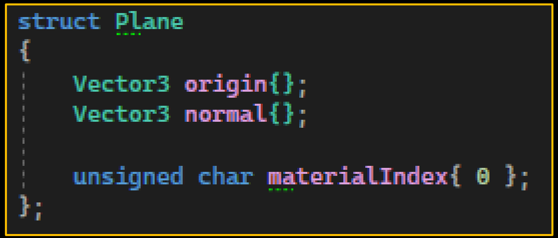
Ray Tracing: Intersections – Ray-Plane?

- An infinite plane can be defined by the following implicit equation:

$$(\boldsymbol{p} - \boldsymbol{o}_{\boldsymbol{plane}}) \cdot \boldsymbol{n} = \boldsymbol{0}$$

- This equation basically says, “the vector from $\boldsymbol{o}_{\boldsymbol{plane}}$ to $\boldsymbol{p}$ is perpendicular to the plane normal”.





```
struct Plane
{
    Vector3 origin{};
    Vector3 normal{};

    unsigned char materialIndex{ 0 };
};
```

Ray Tracing: Intersections – Ray-Plane?

- An infinite plane can be defined by the following implicit equation:

$$(\boldsymbol{p} - \boldsymbol{o}_{\boldsymbol{plane}}) \cdot \boldsymbol{n} = \mathbf{0}$$

- This equation basically says "the vector from $\boldsymbol{o}_{\boldsymbol{plane}}$ to $\boldsymbol{p}$ is perpendicular to the plane normal". So let's plug in our ray equation, and solve it for the unknown $\boldsymbol{t}$ (asking at which interval $\boldsymbol{t}$ do you intersect:

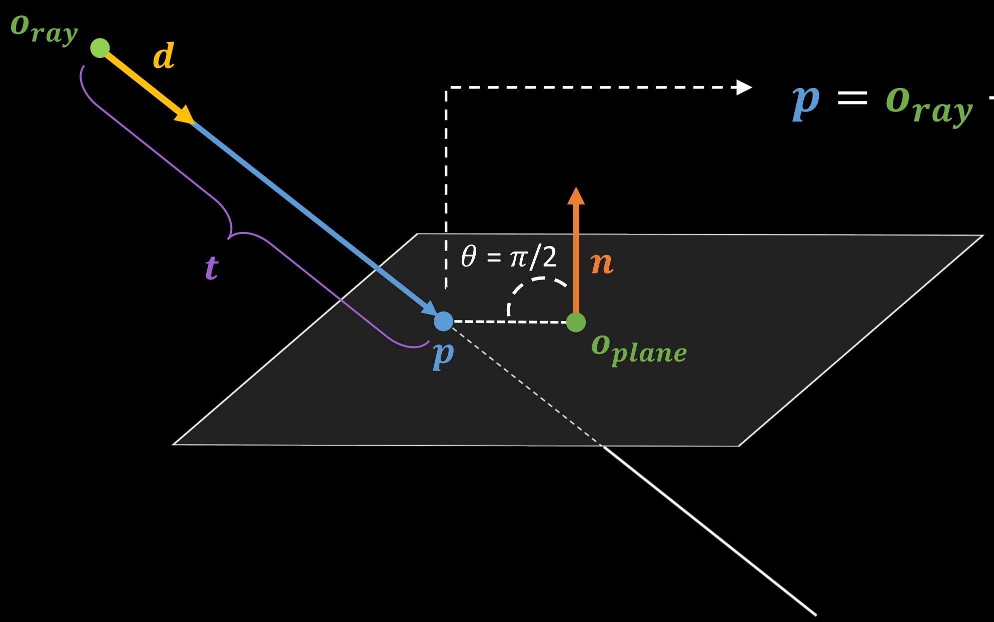
$$\boldsymbol{p}(t) = \boldsymbol{o}_{\boldsymbol{ray}} + t\boldsymbol{d}$$

$$(\boldsymbol{o}_{\boldsymbol{ray}} + \boldsymbol{td} - \boldsymbol{o}_{\boldsymbol{plane}}) \cdot \boldsymbol{n} = \mathbf{0} \quad \rightarrow \quad \boldsymbol{t} = \frac{(\boldsymbol{o}_{\boldsymbol{plane}} - \boldsymbol{o}_{\boldsymbol{ray}}) \cdot \boldsymbol{n}}{\boldsymbol{d} \cdot \boldsymbol{n}}$$

- Just looking at the equation might be confusing but looking at it from a geometrical point of view makes our lives easier!

Ray Tracing: Intersections – Ray-Plane?

- Substituting t in: $p = o_{ray} + td$ when $t = \frac{(o_{plane} - o_{ray}) \cdot n}{d \cdot n}$



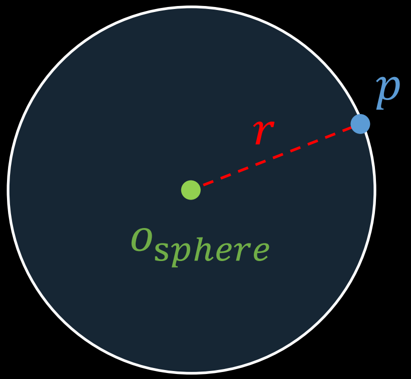
$$p = o_{ray} + \left(\frac{(o_{plane} - o_{ray}) \cdot n}{d \cdot n}\right) d$$

- Once you've found t, don't forget to check if it is between the interval ([tMin, tMax)) before calculating the exact point!

Ray Tracing: Intersections – Ray-Sphere?

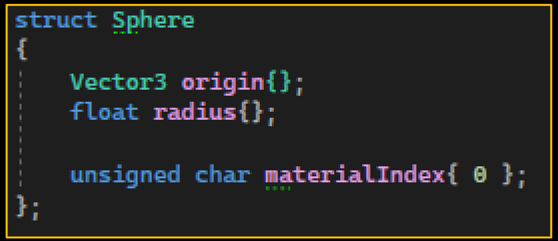
- A sphere (centered at the origin) can be defined by the following implicit equation:

$$\left\| p - o_{sphere} \right\|^2 = r^2$$



$$\downarrow$$

$$(p - o_{\text{sphere}}) \cdot (p - o_{\text{sphere}}) = r^2$$



```
struct Sphere
{
    Vector3 origin{};
    float radius{};

    unsigned char materialIndex{ 0 };
};
```

Ray Tracing: Intersections – Ray-Sphere?

- Let us again replace $\boldsymbol{p}$ with the definition of a ray:

$$(\boldsymbol{p} - \boldsymbol{o}_{\mathbf{sphere}}) \cdot (\boldsymbol{p} - \boldsymbol{o}_{\mathbf{sphere}}) = \boldsymbol{r}^{\mathbf{2}}$$

$$\downarrow$$

$$(\boldsymbol{o_{ray}} + \boldsymbol{td} - \boldsymbol{o_{sphere}}) \cdot (\boldsymbol{o_{ray}} + \boldsymbol{td} - \boldsymbol{o_{sphere}}) - \boldsymbol{r}^{\mathbf{2}} = \mathbf{0}$$

- Expanding and moving the term around, we get:

$$\boldsymbol{t}^{\mathbf{2}}\underbrace{(\boldsymbol{d} \cdot \boldsymbol{d})}_{\boldsymbol{A}} + \boldsymbol{t}\underbrace{\left(\mathbf{2}\boldsymbol{d} \cdot (\boldsymbol{o_{ray}} - \boldsymbol{o_{sphere}})\right)}_{\boldsymbol{B}} + \underbrace{\left((\boldsymbol{o_{ray}} - \boldsymbol{o_{sphere}}) \cdot (\boldsymbol{o_{ray}} - \boldsymbol{o_{sphere}})\right) - \boldsymbol{r}^{\mathbf{2}}}_{\boldsymbol{C}} = \mathbf{0}$$

$$\downarrow$$

$$\boldsymbol{At}^{\mathbf{2}} + \boldsymbol{Bt} + \boldsymbol{C} = \mathbf{0}$$

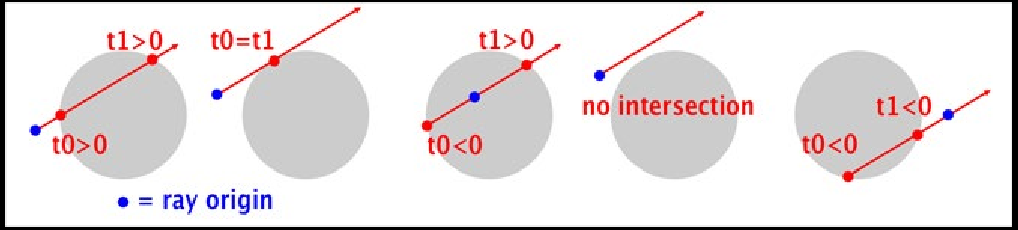
Ray Tracing: Intersections – Ray-Sphere?

- If we solve this quadratic equation for t, using the parameters $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$, we get:

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

- $B^2 - 4AC$ is the discriminant of the equation, and
  - if discriminant < 0: ray does not intersect.
  - if discriminant $== 0$: ray just touches the sphere in one point (tangent).
  - if discriminant > 0: ray does intersect in two points.

- We are only interested in full intersection (so discriminant > 0). If this happens, we need to again check if it's between the interval ([tMin, tMax)). If so, calculate the point using:



1

$\boxed{?}$

1 ScratchAPixel

Ray Tracing: Intersections – Ray-Sphere?

- If we solve this quadratic equation for t, using the parameters A, B and C , we get:

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

- $B^2 - 4AC$ is the discriminant of the equation, and
  - if discriminant < 0: ray does not intersect.
  - if discriminant == 0: ray just touches the sphere in one point (tangent).
  - if discriminant > 0: ray does intersect in two points.

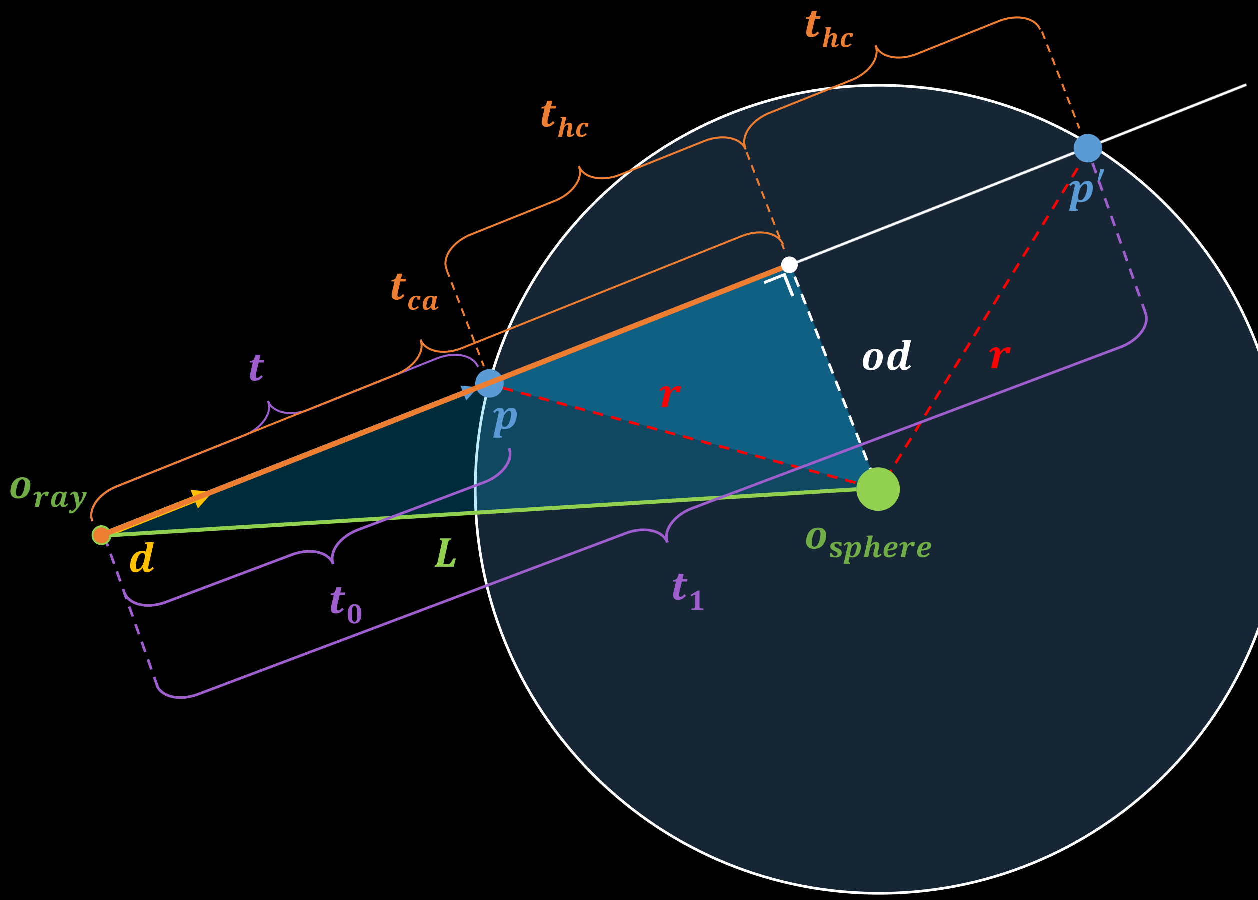
- We are only interested in full intersection (so discriminant > 0). If this happens, we need to again check if it's between the interval ([tMin, tMax)). If so, calculate the point using:

$$p = o_{ray} + \left(\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}\right) d$$

Use subtraction, except when t < tMin, than use addition for t.

Ray Tracing: Intersections – Ray-Sphere

- That was the analytic solution, what about the geometric solution:



$$L = o_{sphere} - o_{ray}$$

$$t_{ca} = L \cdot d$$

opposite side^2 + adjacent side^2 = hypotenuse^2

$$od = \sqrt{L^2 - t_{ca}^2}$$

$$\downarrow$$

$$od = \sqrt{L \cdot L - t_{ca} \cdot t_{ca}}$$

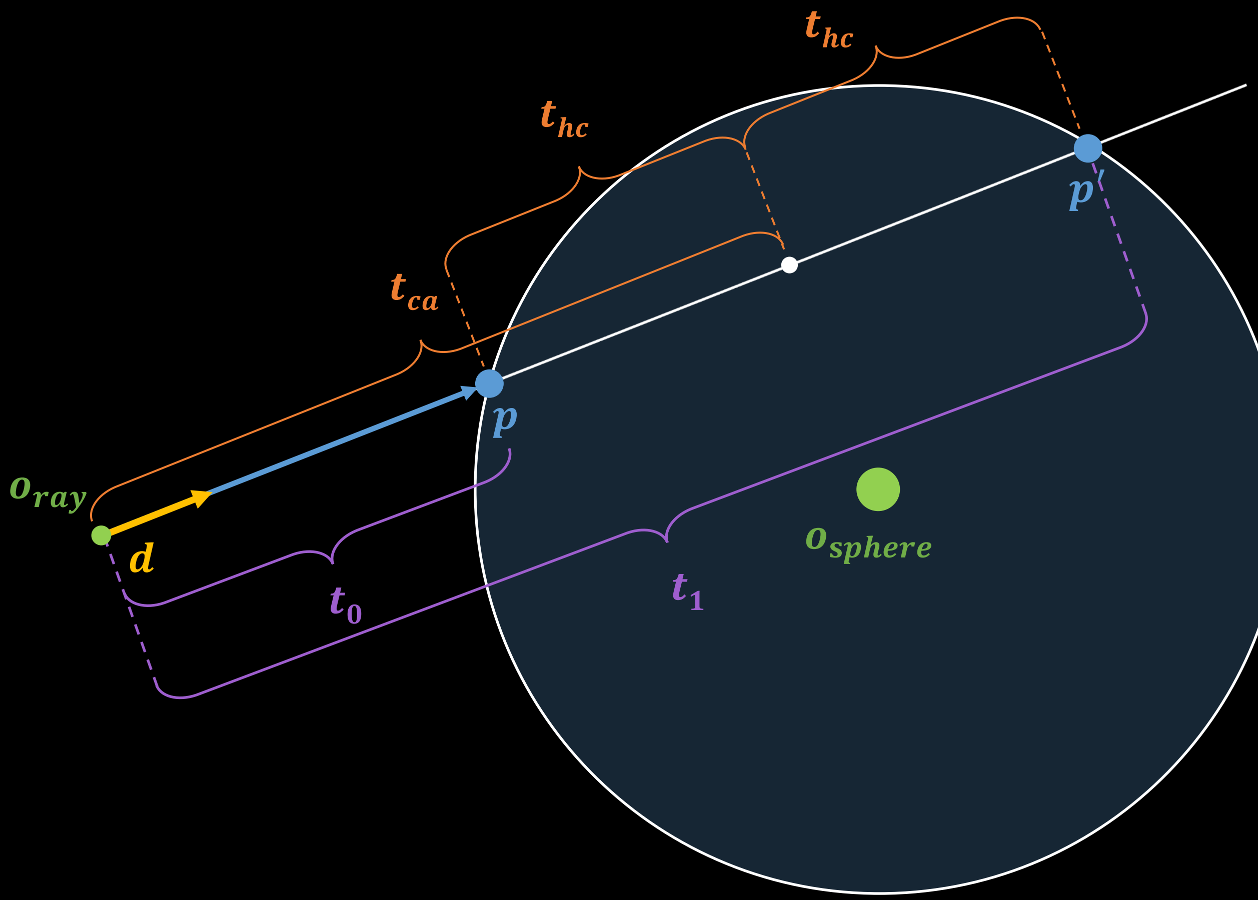
$$t_{hc} = \sqrt{r^2 - od^2}$$

$$t_0 = t_{ca} - t_{hc}$$

$$t_1 = t_{ca} + t_{hc}$$

Ray Tracing: Intersections – Ray-Sphere

- That was the analytic solution, what about the geometric solution:



$$L = o_{sphere} - o_{ray}$$

$$t_{ca} = L \cdot d$$

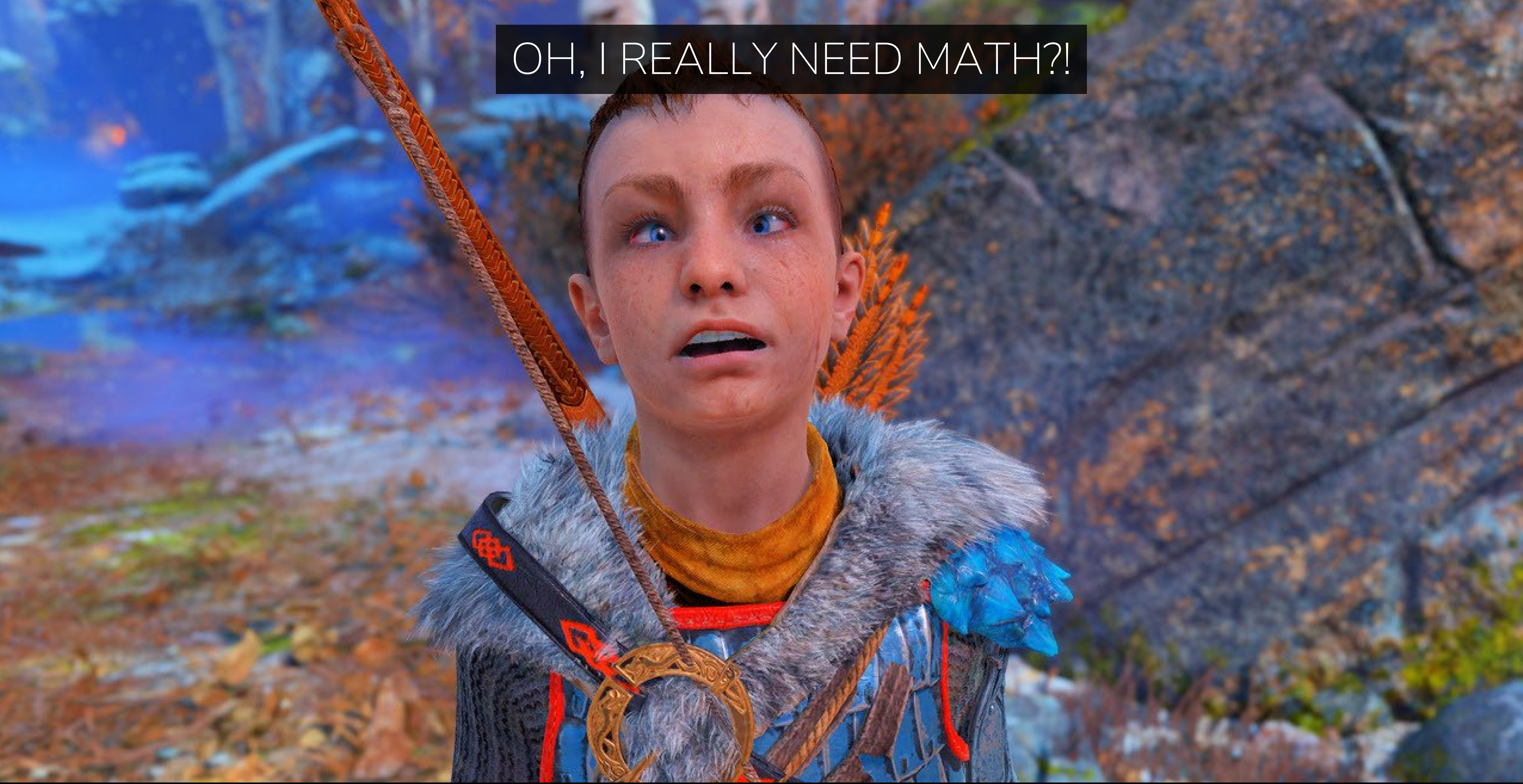
$$od^2 = \|Reject(L, d)\|^2$$

$$t_{hc} = \sqrt{r^2 - od^2}$$

$$t_0 = t_{ca} - t_{hc}$$

$$t_1 = t_{ca} + t_{hc}$$

$$p = o_{ray} + t_0 d$$



OH, I REALLY NEED MATH?!

Ray Tracing: What to do?

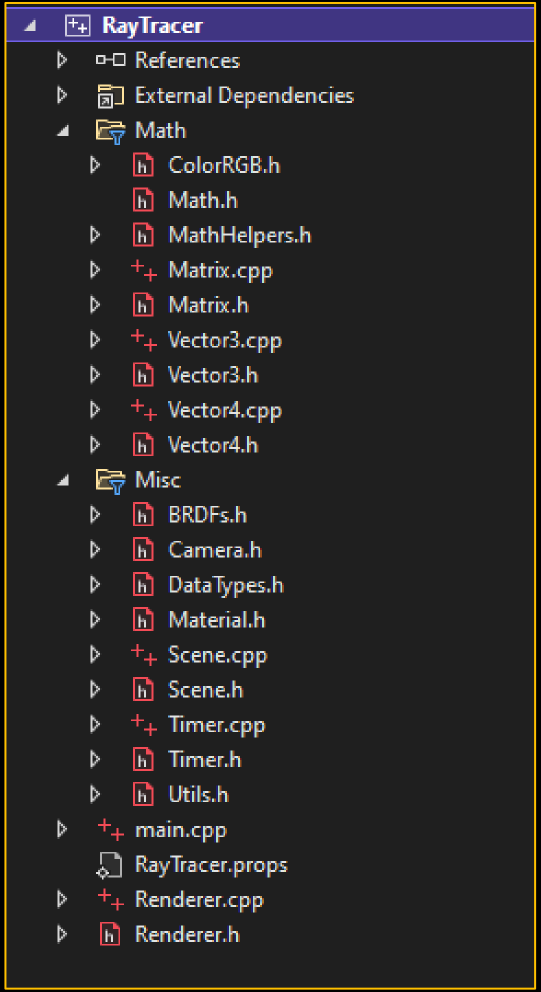
- You can find a start project on ‘leho’. It contains the window setup, a small render loop and some math files. It’s your task to build your own ray tracer (using this start project as the foundation)!
- You’ll work on this project every week. Every week you are going to add extra features.
- It’s up to YOU to structure your code, add extra files and/or expand functionality. Do NOT change the math files though! If an important feature is missing, please notify your teacher!
- Code quality is important! So make sure you write clean code! We’ll also pay attention to:
  - Coding Standards
  - Containers (do you use the correct containers, etc.)
  - Avoiding unnecessary copies (passing by reference vs by value, std::move, etc.)
  - Performance (avoid unnecessary calculations, sqrt, references, etc.)
  - Features (are they implemented correctly, extra features, etc.)
  - Code architecture & design (do you split into classes/structs/functions, logical filter structure, etc.)
- Let’s take a look at the start project…

Ray Tracing: Constraints

- Row-Based Matrices and Vectors
- Left-Handed System! (Z-axis points into the screen)
- Pixel coordinates in Raster Space
- Calculate colors in [0, 1] range (even though rendering is done in [0, 255])
- … more to follow …

Project not conform constraints → 0/20!

GP1 Raytracer Project



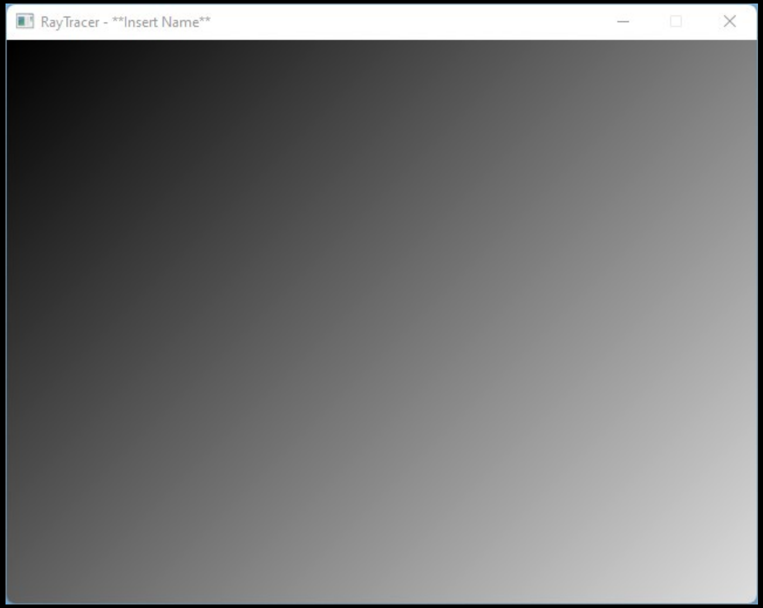
```
◢ RayTracer
    ▷ References
    ▷ External Dependencies
    ◢ Math
        ▷ ColorRGB.h
          Math.h
        ▷ MathHelpers.h
        ▷ Matrix.cpp
        ▷ Matrix.h
        ▷ Vector3.cpp
        ▷ Vector3.h
        ▷ Vector4.cpp
        ▷ Vector4.h
    ◢ Misc
        ▷ BRDFs.h
        ▷ Camera.h
        ▷ DataTypes.h
        ▷ Material.h
        ▷ Scene.cpp
        ▷ Scene.h
        ▷ Timer.cpp
        ▷ Timer.h
        ▷ Utils.h
    ▷ main.cpp
      RayTracer.props
    ▷ Renderer.cpp
    ▷ Renderer.h
```

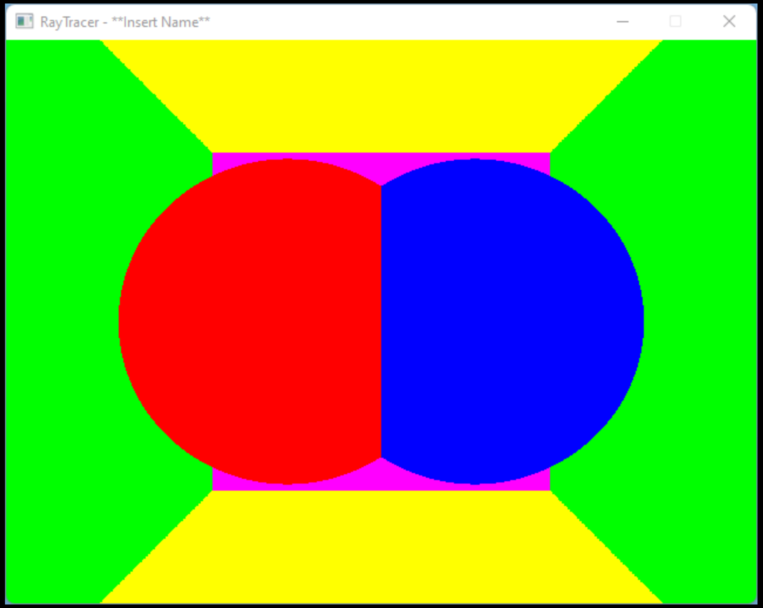
- Should compile out-of-the-box
- Skeleton with (empty) functions to create a 'basic' Raytracer

- Math Filter > All Math related objects
- Misc
  - BRDFs > BRDF (W3) helper functions
  - Camera > Basic Camera (W2)
  - DataTypes > File containing POD structs (geometries, lights, ray, …)
  - Material > Materials used to shade our objects
  - Scene > Simple scene that keeps track of the camera, geometries & lights
  - Timer > Simple timer class with access to elapsed and total time
  - Utils > Utility functions
    - GeometryUtils > Various geometry hit-test functions
    - LightUtils > Helpers functions for different light types
    - Utils > General utility functions
  - Renderer > Where the magic happens! (Screen update & Calculations)
  - main > Program main entrypoint

W1 Lab – Objective

- Objective = Raytrace spheres & planes using a fixed 'camera' and a simple material



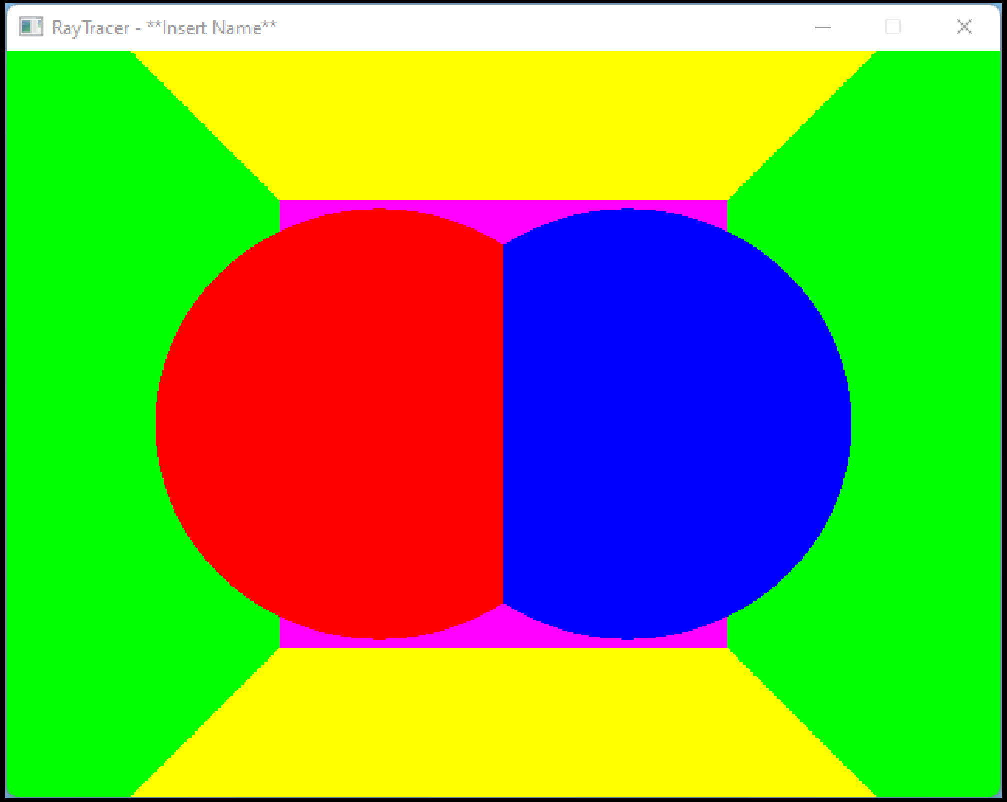


- Most of the function in the start project are still empty – It is your job to complete these and use them where appropriate. The following slides contains some implementation instructions/steps that will help you to implement the objective for this lab.

W1 Lab – Objective

- Raytracing Flow (W1)

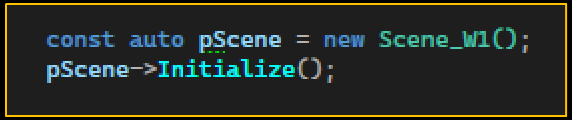
```
For Each Pixel (Px / Py)
    Calculate 'viewRay'
    Scene::GetClosestHit (from Ray)
        Iterate Sphere Geometries
            Hit? Closest Hit?
        Iterate Plane Geometries
            Hit? Closest Hit?
    Did Hit?
        NO > Pixel = BLACK
        YES > Pixel = Material Color
```



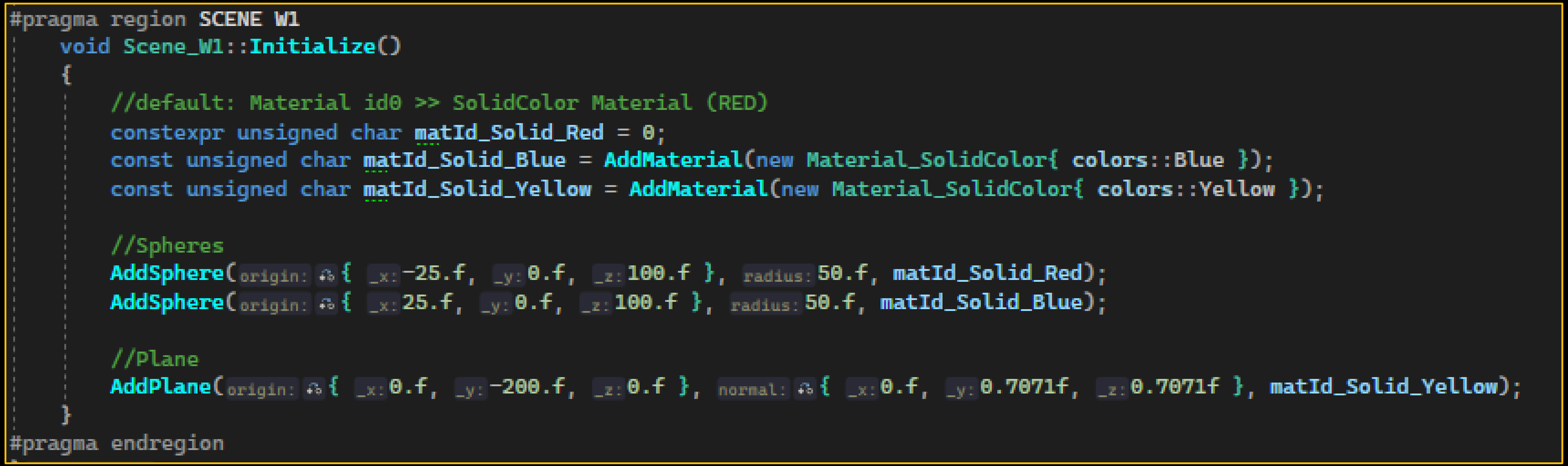
W1 Lab – Todo (1)

Make sure ‘Scene_W1’ is the active scene (should already be the case)

- Describes a scene with two intersecting spheres (Red/Blue) and some planes (Green/Yellow/Magenta)
- Main.cpp



```cpp
const auto pScene = new Scene_W1();
pScene->Initialize();
```



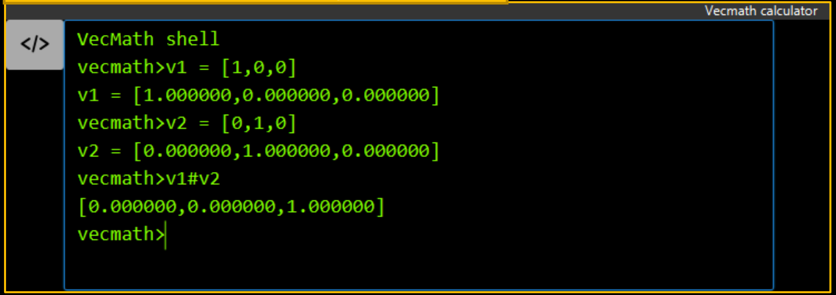
```cpp
#pragma region SCENE W1
    void Scene_W1::Initialize()
    {
        //default: Material id0 >> SolidColor Material (RED)
        constexpr unsigned char matId_Solid_Red = 0;
        const unsigned char matId_Solid_Blue = AddMaterial(new Material_SolidColor{ colors::Blue });
        const unsigned char matId_Solid_Yellow = AddMaterial(new Material_SolidColor{ colors::Yellow });

        //Spheres
        AddSphere(origin: { _x:-25.f, _y:0.f, _z:100.f }, radius:50.f, matId_Solid_Red);
        AddSphere(origin: { _x:25.f, _y:0.f, _z:100.f }, radius:50.f, matId_Solid_Blue);

        //Plane
        AddPlane(origin: { _x:0.f, _y:-200.f, _z:0.f }, normal: { _x:0.f, _y:0.7071f, _z:0.7071f }, matId_Solid_Yellow);
    }
#pragma endregion
```

W1 Lab – Todo (2)

Implement Vector3::Dot/Cross & Vector4::Dot (Test your implementation)

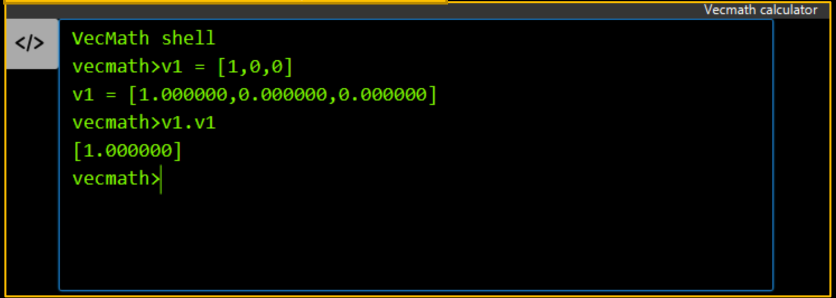
- Alternatively, you can use the VecMath module inside FxMath

VecMath Cross ('#' operator)



```
VecMath shell
vecmath>v1 = [1,0,0]
v1 = [1.000000,0.000000,0.000000]
vecmath>v2 = [0,1,0]
v2 = [0.000000,1.000000,0.000000]
vecmath>v1#v2
[0.000000,0.000000,1.000000]
vecmath>
```

VecMath Dot ('.' Operator)



```
VecMath shell
vecmath>v1 = [1,0,0]
v1 = [1.000000,0.000000,0.000000]
vecmath>v1.v1
[1.000000]
vecmath>
```

In Code

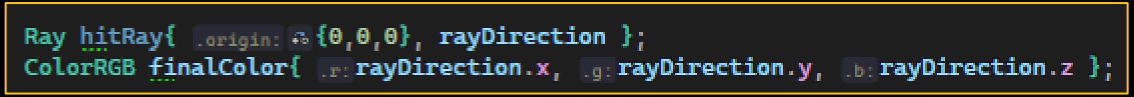
```
float dotResult{};
dotResult = Vector3::Dot(Vector3::UnitX, Vector3::UnitX); //(1) Same Direction
dotResult = Vector3::Dot(Vector3::UnitX, -Vector3::UnitX); //(-1) Opposite Direction
dotResult = Vector3::Dot(Vector3::UnitX, Vector3::UnitY); //(0) Perpendicular

Vector3 crossResult{}; //Left-Handed!
crossResult = Vector3::Cross(Vector3::UnitZ, Vector3::UnitX); //(0,1,0) UnitY
crossResult = Vector3::Cross(Vector3::UnitX, Vector3::UnitZ); //(0,-1,0) -UnitY
```

W1 Lab – Todo (3)

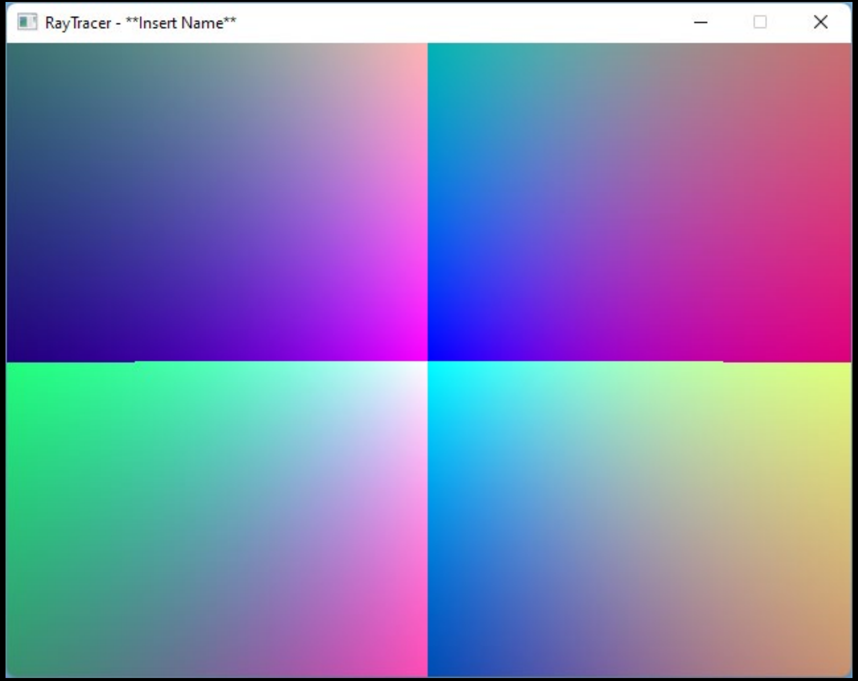
Calculate Ray for each pixel of the screen

- Verify your solution by outputting a color for the pixel based on the ray direction (see below)

- Check FxMath
  - Raster Space
  - Camera Concept
  - From Raster Space to …
  - From Camera Space to…



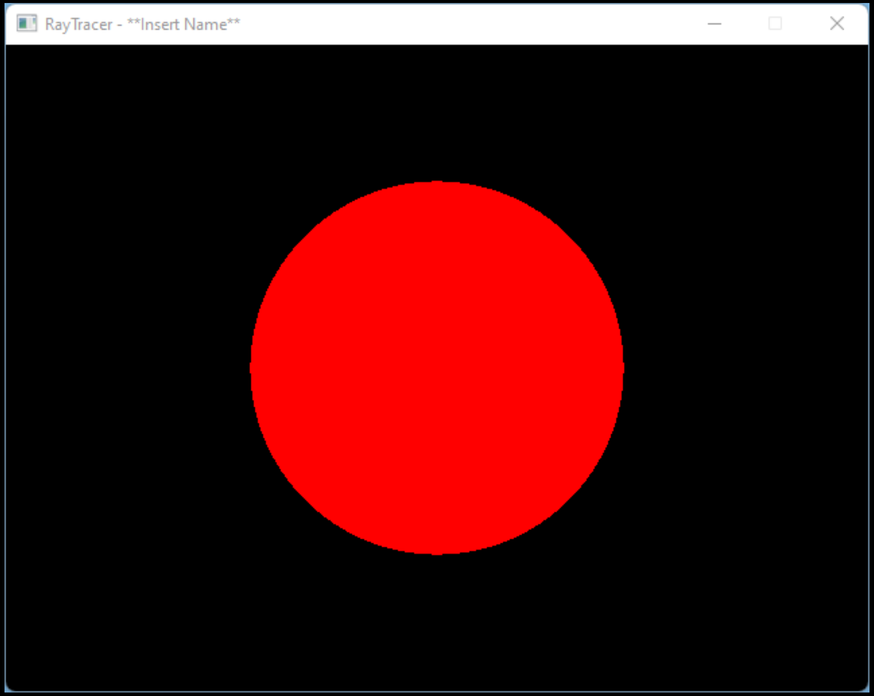
```
Ray hitRay{ .origin: {0,0,0}, rayDirection };
ColorRGB finalColor{ .r: rayDirection.x, .g: rayDirection.y, .b: rayDirection.z };
```

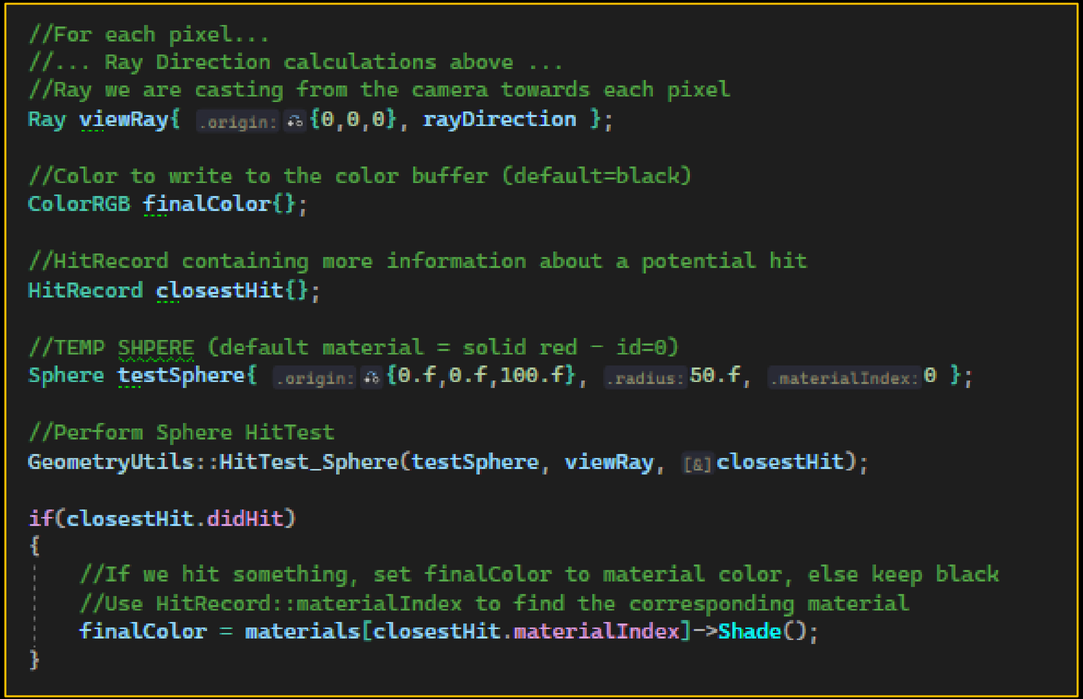


W1 Lab – Todo (4)

Implement Sphere HitTest function

- GeometryUtils::HitTest_Sphere(...) @ Utils.h
- For now, verify your implementation by RayTracing a temporary sphere (see below)



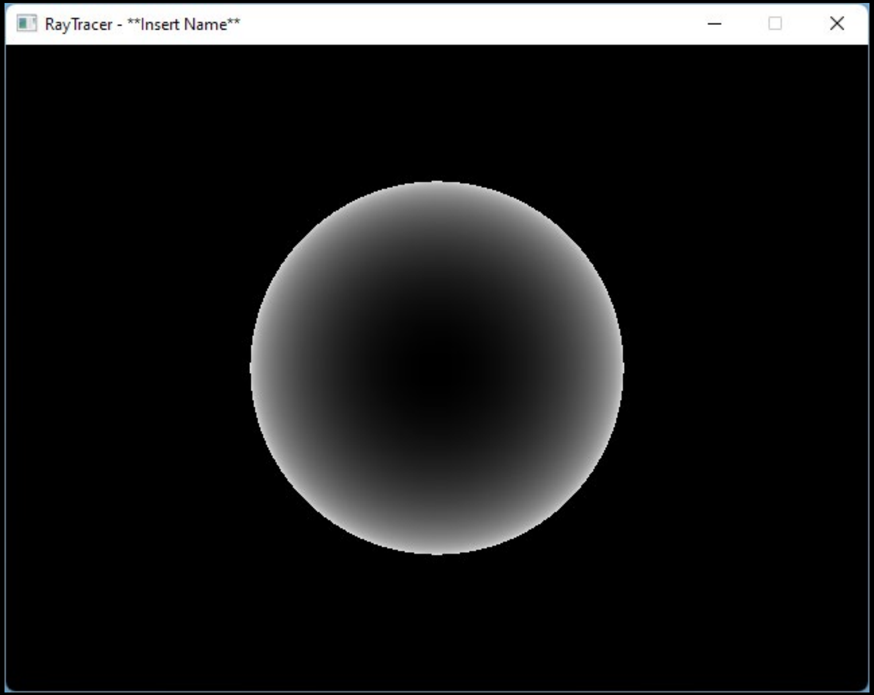


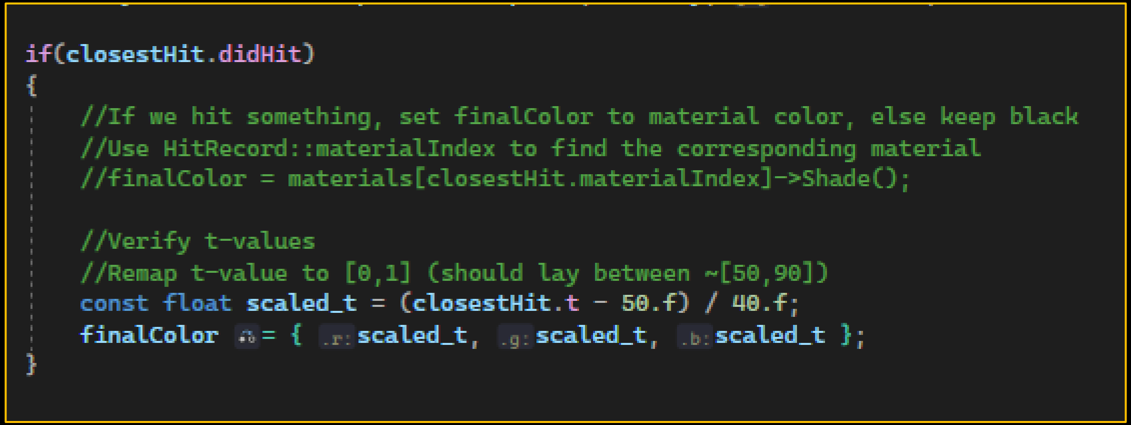
```
//For each pixel...
//... Ray Direction calculations above ...
//Ray we are casting from the camera towards each pixel
Ray viewRay{ .origin: {0,0,0}, rayDirection };

//Color to write to the color buffer (default=black)
ColorRGB finalColor{};

//HitRecord containing more information about a potential hit
HitRecord closestHit{};

//TEMP SHPERE (default material = solid red - id=0)
Sphere testSphere{ .origin: {0.f,0.f,100.f}, .radius: 50.f, .materialIndex: 0 };

//Perform Sphere HitTest
GeometryUtils::HitTest_Sphere(testSphere, viewRay, [&] closestHit);

if(closestHit.didHit)
{
    //If we hit something, set finalColor to material color, else keep black
    //Use HitRecord::materialIndex to find the corresponding material
    finalColor = materials[closestHit.materialIndex]->Shade();
}
```

W1 Lab – Todo (5)

Implement Sphere HitTest function

- Also, verify your calculated t-values (closestHit.t)



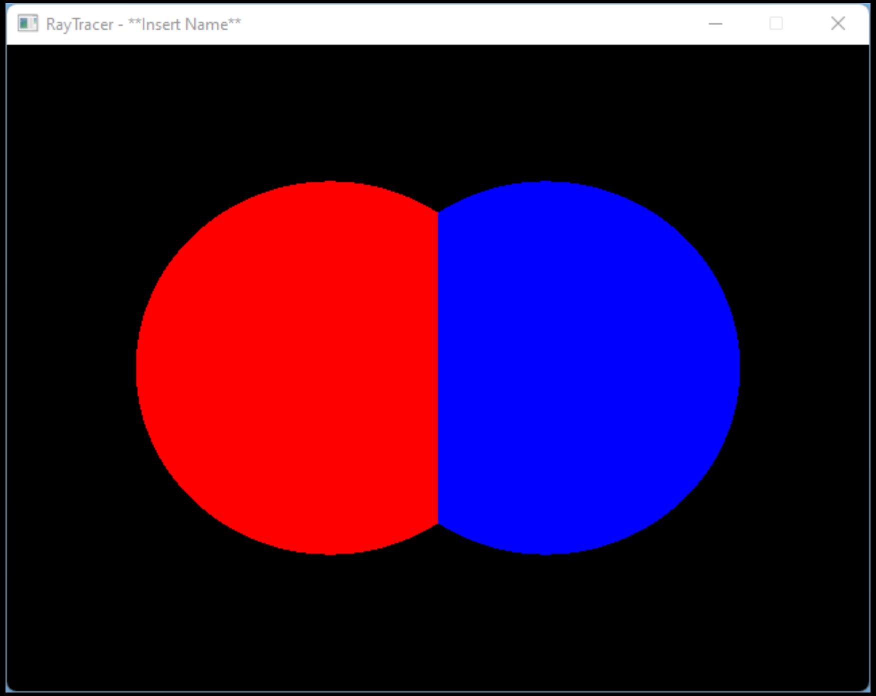


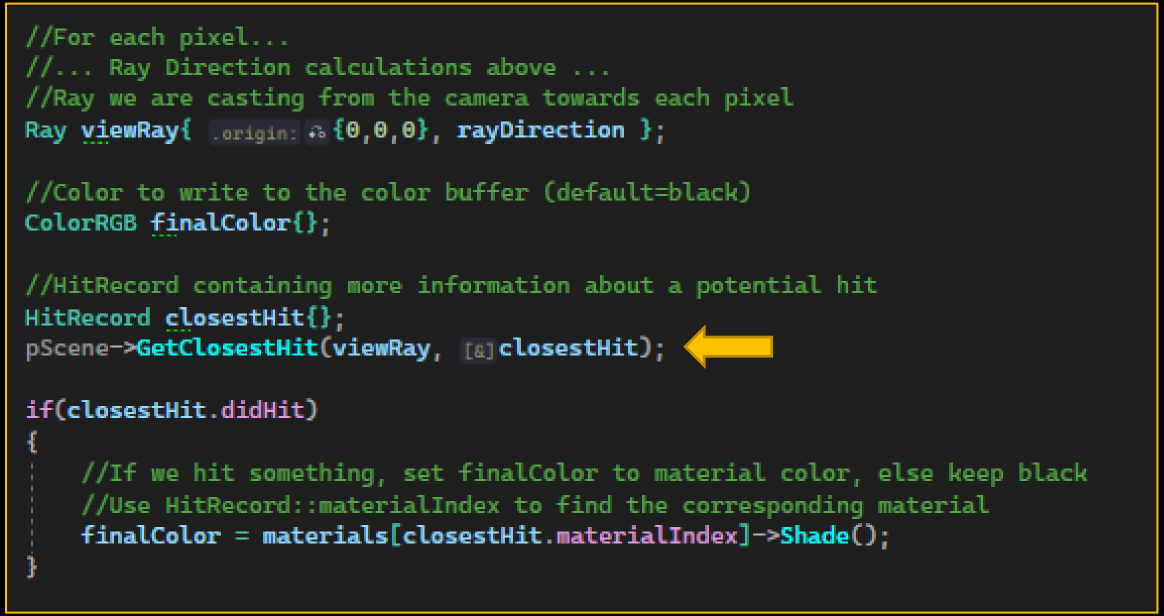
```cpp
if(closestHit.didHit)
{
    //If we hit something, set finalColor to material color, else keep black
    //Use HitRecord::materialIndex to find the corresponding material
    //finalColor = materials[closestHit.materialIndex]->Shade();

    //Verify t-values
    //Remap t-value to [0,1] (should lay between ~[50,90])
    const float scaled_t = (closestHit.t - 50.f) / 40.f;
    finalColor = { scaled_t, scaled_t, scaled_t };
}
```

W1 Lab – Todo (6)

Implement Scene::GetClosestHit function (Spheres only)

- This function iterates all spheres (planes, triangles are added later) and returns the HitRecord of the closest (smallest t-value) hit
- Remove the previous test code, and instead call this function to get the closestHit.
  Spheres defined in the W1_Scene should be visible now – verify that both spheres are intersecting correctly!



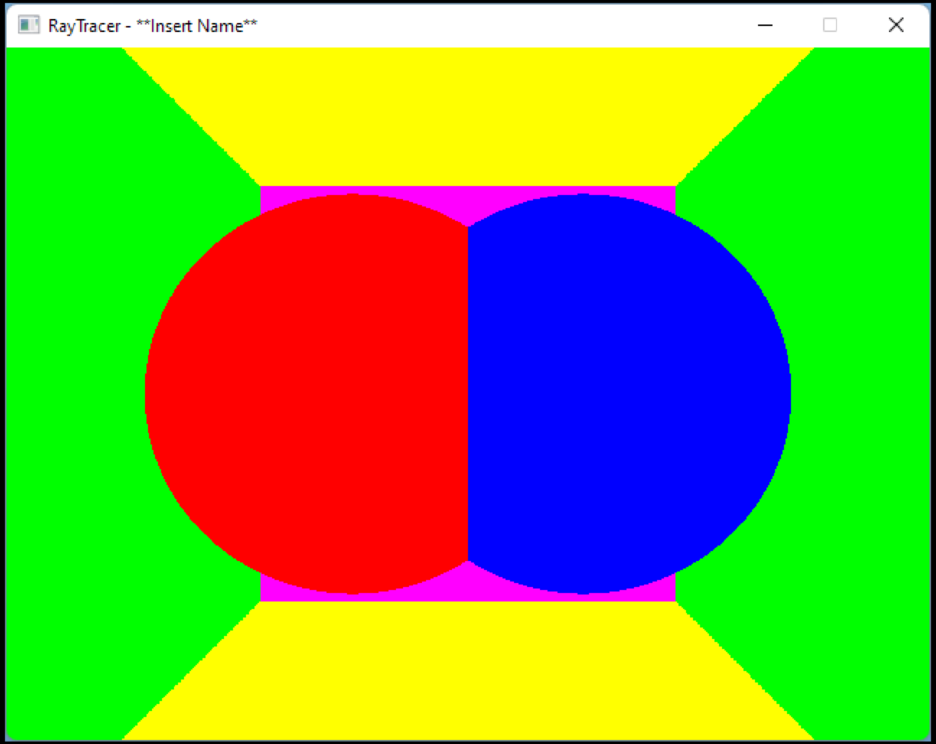


```
//For each pixel...
//... Ray Direction calculations above ...
//Ray we are casting from the camera towards each pixel
Ray viewRay{ .origin: {0,0,0}, rayDirection };

//Color to write to the color buffer (default=black)
ColorRGB finalColor{};

//HitRecord containing more information about a potential hit
HitRecord closestHit{};
pScene->GetClosestHit(viewRay, [&]closestHit);

if(closestHit.didHit)
{
    //If we hit something, set finalColor to material color, else keep black
    //Use HitRecord::materialIndex to find the corresponding material
    finalColor = materials[closestHit.materialIndex]->Shade();
}
```

W1 Lab – Todo (7)

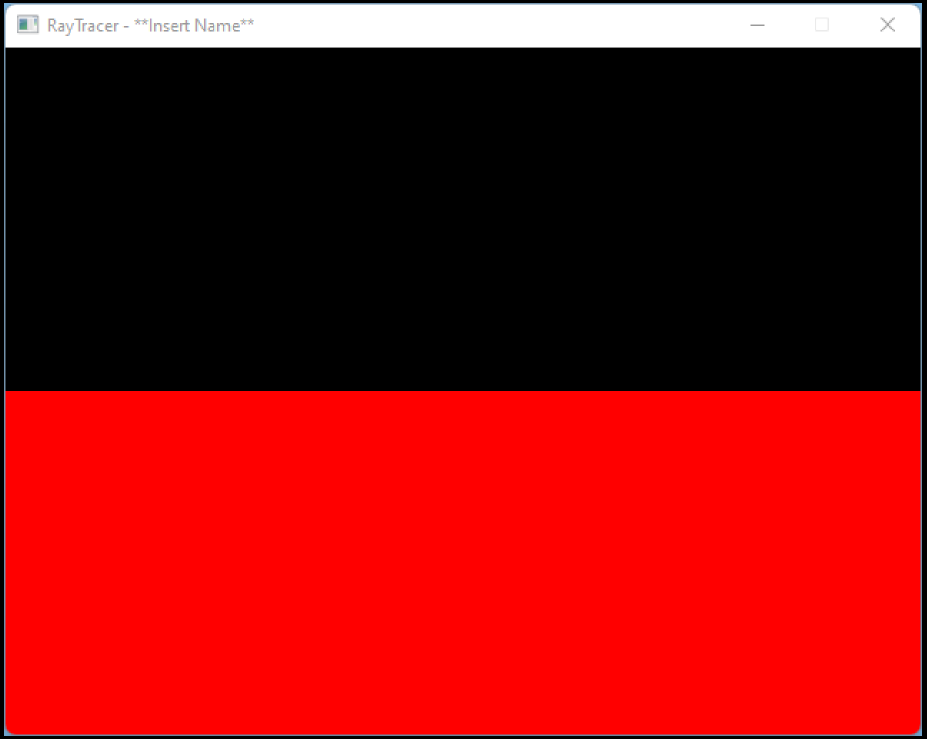
Apply the same logic (~spheres) to plane geometries

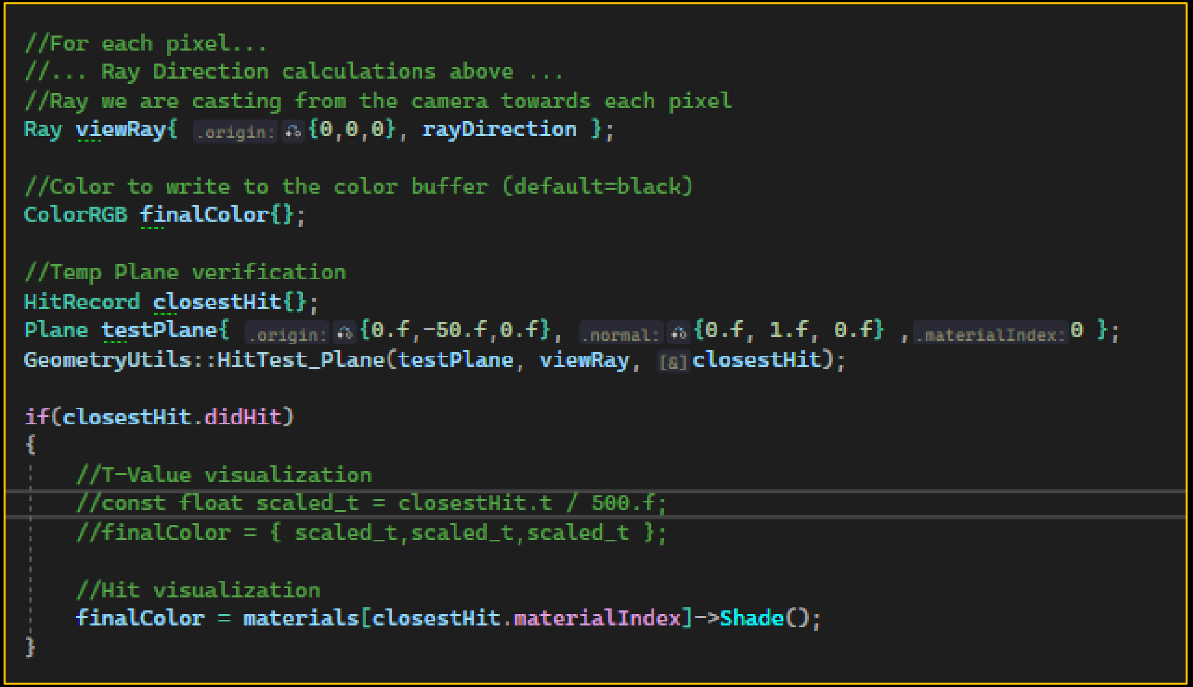
- Implement GeometryUtils::HitTest_Plane(...) @ Utils.h
- Take them into account for Scene::GetClosestHit(...)



W1 Lab – Todo (9)

Extra > In case you want to verify your Plane Hit implementation (hit)

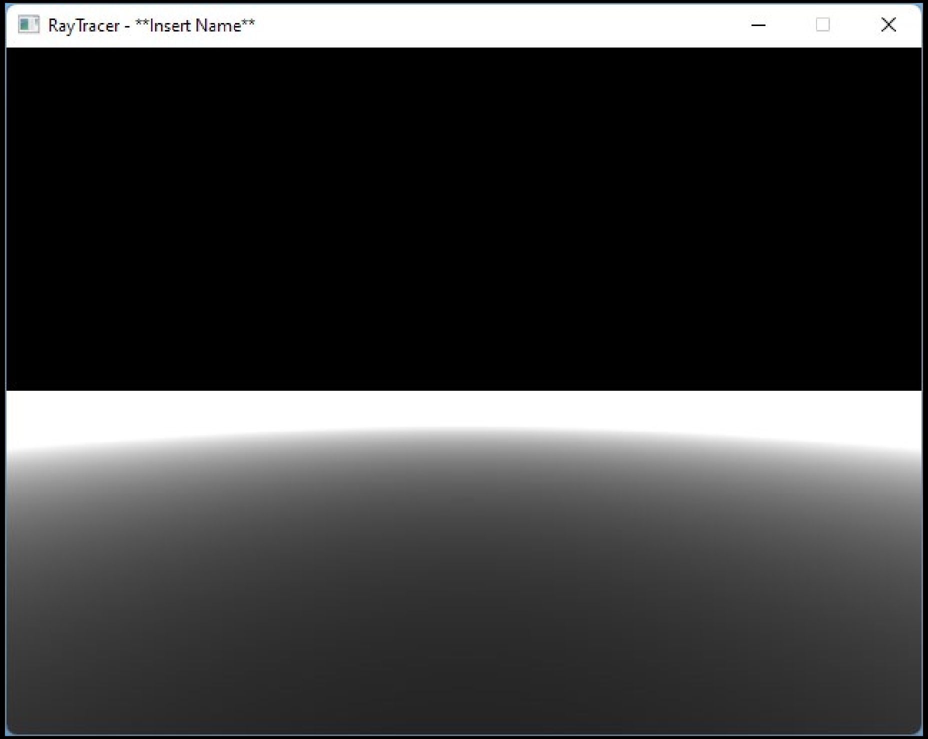


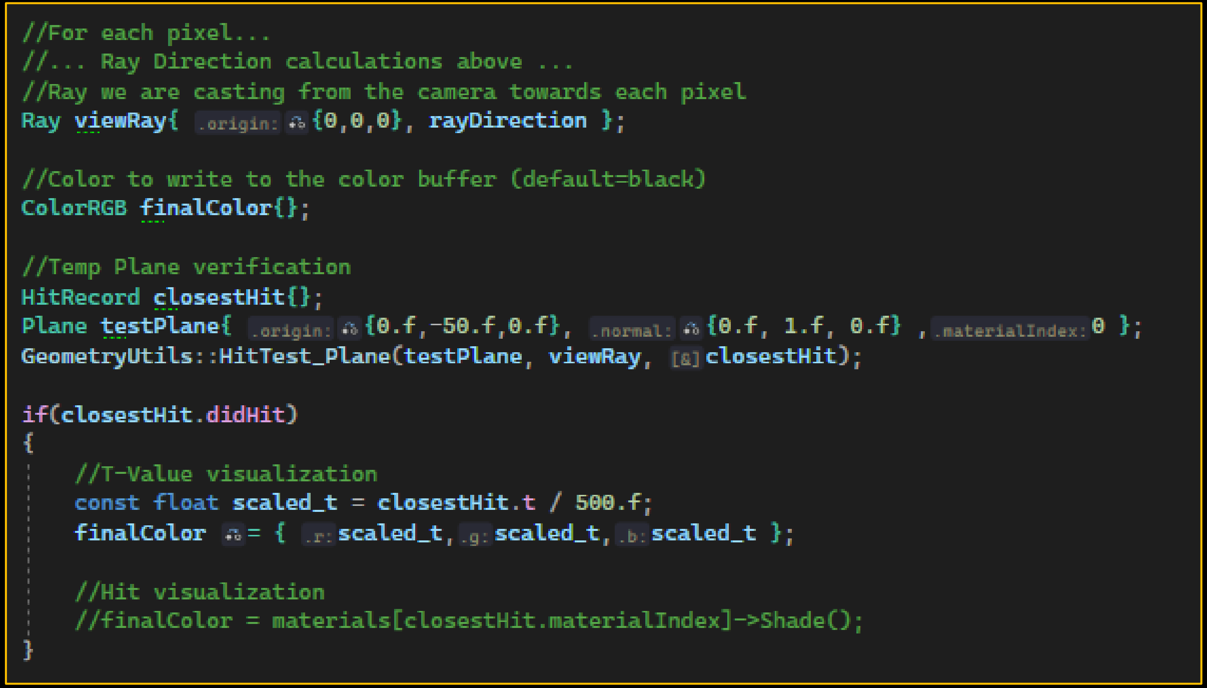


```cpp
//For each pixel...
//... Ray Direction calculations above ...
//Ray we are casting from the camera towards each pixel
Ray viewRay{ .origin: {0,0,0}, rayDirection };

//Color to write to the color buffer (default=black)
ColorRGB finalColor{};

//Temp Plane verification
HitRecord closestHit{};
Plane testPlane{ .origin: {0.f,-50.f,0.f}, .normal: {0.f, 1.f, 0.f} , .materialIndex: 0 };
GeometryUtils::HitTest_Plane(testPlane, viewRay, [&] closestHit);

if(closestHit.didHit)
{
    //T-Value visualization
    //const float scaled_t = closestHit.t / 500.f;
    //finalColor = { scaled_t,scaled_t,scaled_t };

    //Hit visualization
    finalColor = materials[closestHit.materialIndex]->Shade();
}
```

W1 Lab – Todo (10)

Extra > In case you want to verify your Plane Hit implementation (t-values)





```
//For each pixel...
//... Ray Direction calculations above ...
//Ray we are casting from the camera towards each pixel
Ray viewRay{ .origin: {0,0,0}, rayDirection };

//Color to write to the color buffer (default=black)
ColorRGB finalColor{};

//Temp Plane verification
HitRecord closestHit{};
Plane testPlane{ .origin: {0.f,-50.f,0.f}, .normal: {0.f, 1.f, 0.f} , .materialIndex: 0 };
GeometryUtils::HitTest_Plane(testPlane, viewRay, [&] closestHit);

if(closestHit.didHit)
{
    //T-Value visualization
    const float scaled_t = closestHit.t / 500.f;
    finalColor = { .r: scaled_t, .g: scaled_t, .b: scaled_t };

    //Hit visualization
    //finalColor = materials[closestHit.materialIndex]->Shade();
}
```

Final Hints

- If stuck, **DEBUG**! → https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints?view=vs-2019
- Make sure you understand the **theory**.
- Make a **"battleplan"** on paper (code design → know your code).
- Write comments and use logical variable names.
- Don't be afraid to **refactor** your code! https://en.wikipedia.org/wiki/Code_refactoring
- **Version control** you code! Submit changes **regularly** (GITHUB)!
  If really stuck, mail your teachers (both) BUT:
  - **DON'T**: "Sir, it doesn't work.", "Sir, my project just crashes.", "Sir, I am lost", etc.
  - **DO**: "Sir, I've working on this feature. It doesn't do what it should do. It renders the sphere like this: *insert* picture. I've narrowed the problem down to this line of code (file + line number). I guess 'x' is wrong? I can't seem to wrap my head around why this is wrong because 'y' seems correct. Here is my project, can you take a look?" → **BE SPECIFIC!**
  - **Do not attach your project to the mail, instead give us a link to your GitHub repo (make sure we can access it!)**
- Test/Run your code in both Debug and **Release** (only x64).

GOOD LUCK!