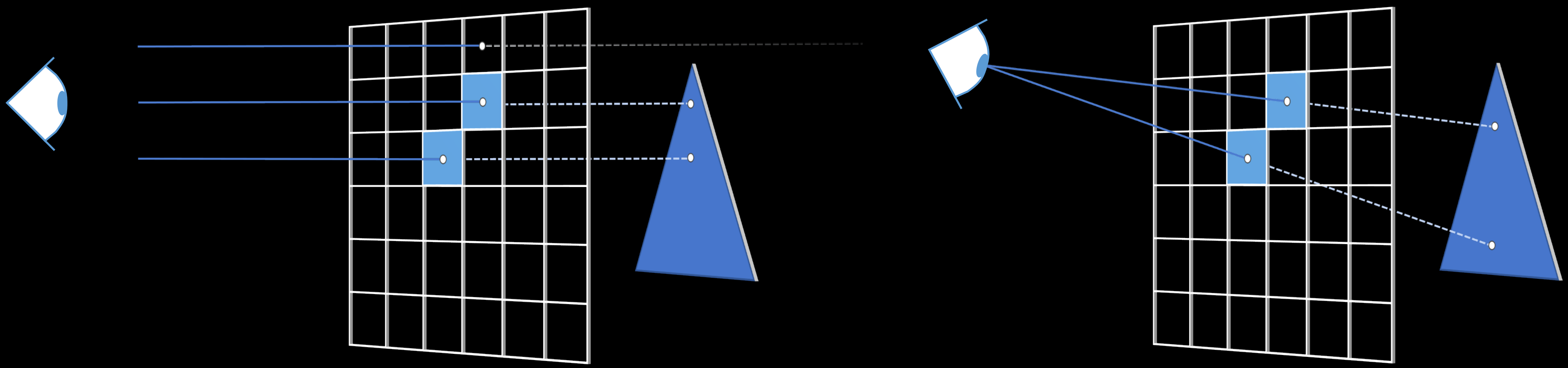
GRAPHICS PROGRAMMING I

PERSPECTIVE CAMERA

Ray Tracing: Orthographic vs Perspective?

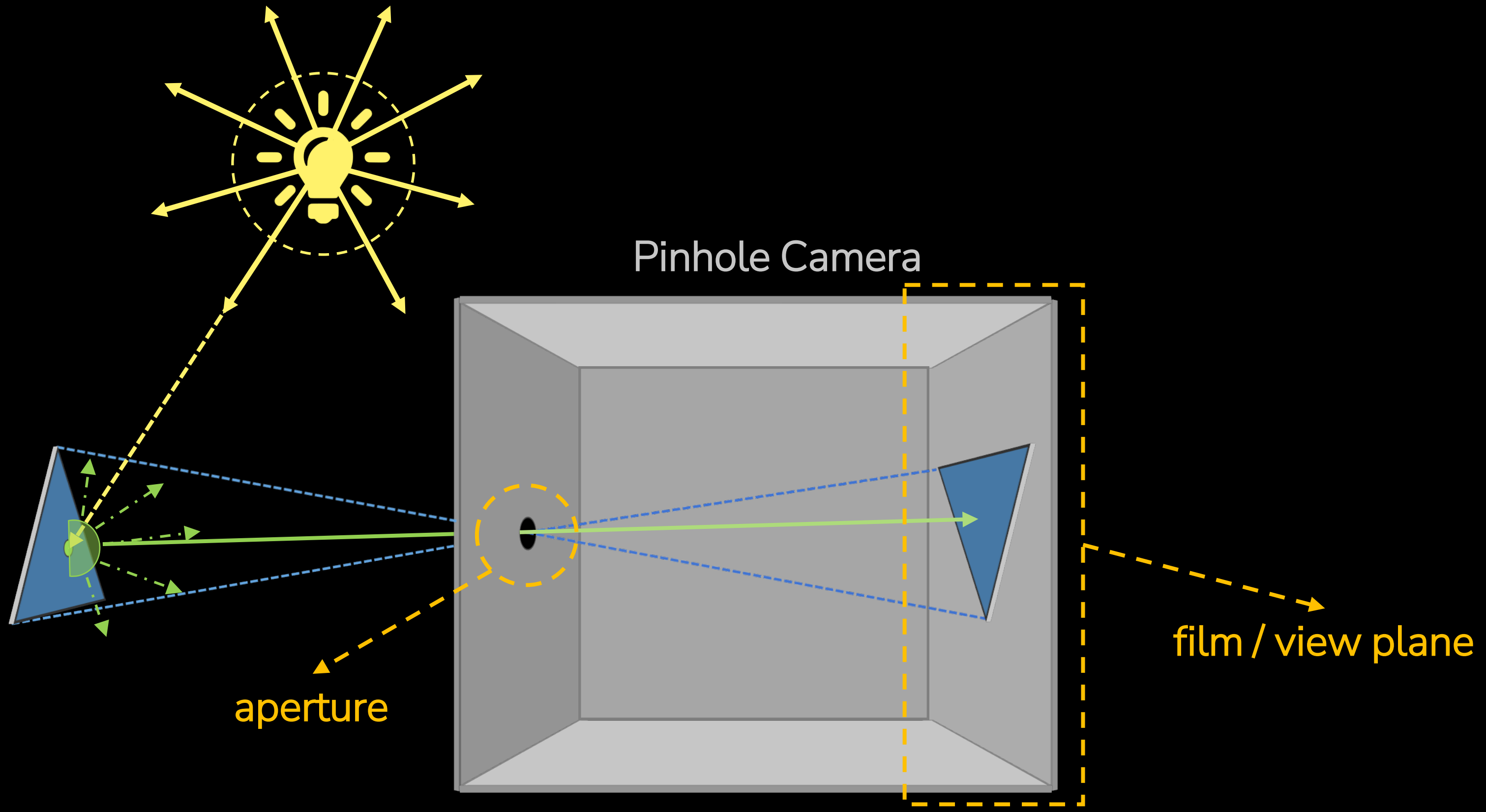
Last week we implemented a first version of the Raytracer. We shoot a ray for every pixel into the virtual world. All these pixels are on an 'imaginary' plane we call the view plane.

- Orthographic Projection > Shooting parallel rays from the camera's origin through the pixel
- Perspective Projection > Shooting rays from the camera's origin through the pixel



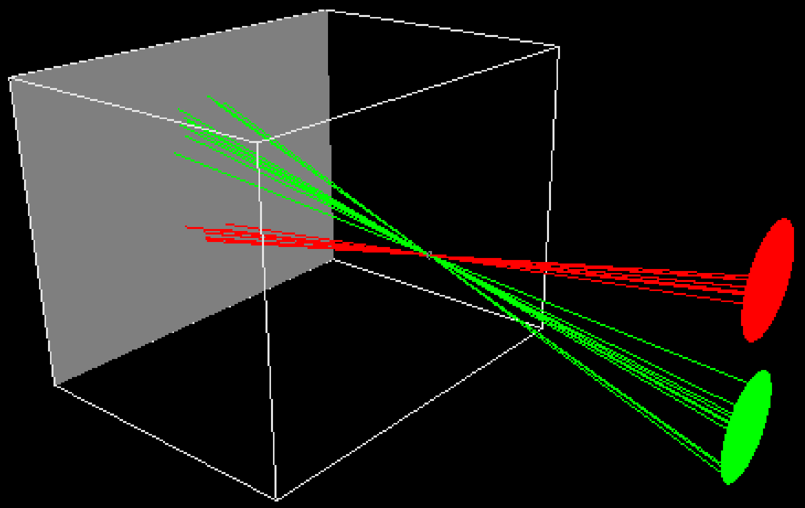
Ray Tracing: Camera Concept

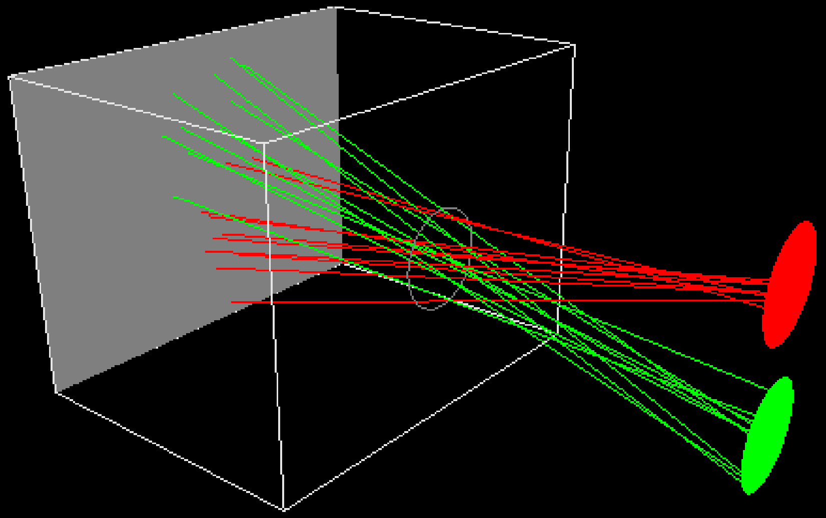
- Let’s take a look at how images are formed in a **camera** and see if we can borrow some ideas.

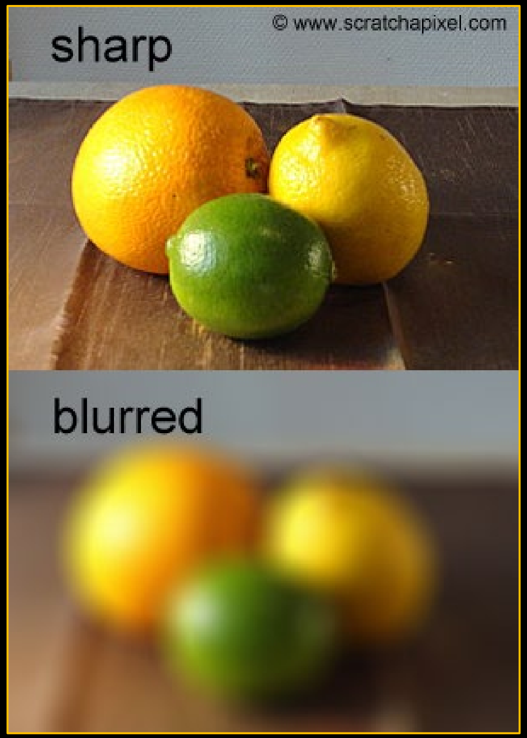


Ray Tracing: Camera Concept

- In a **pinhole** camera, the size of the aperture really matters! To get a sharp image, you want a small aperture so that only a small area of the object is projected onto a single point in the image. In computer graphics this isn't the case because a renderer usually uses an ideal pinhole camera. We use different techniques to create certain effects (e.g. Depth Of Field).

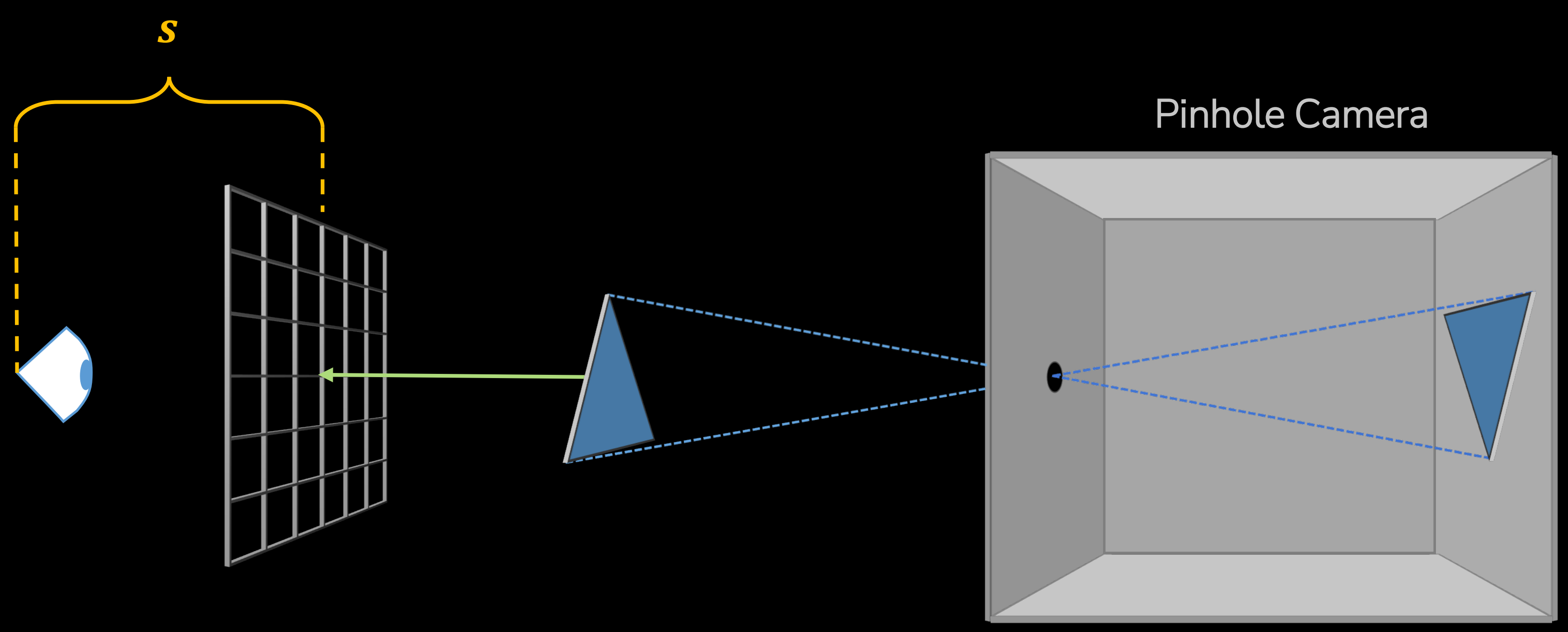






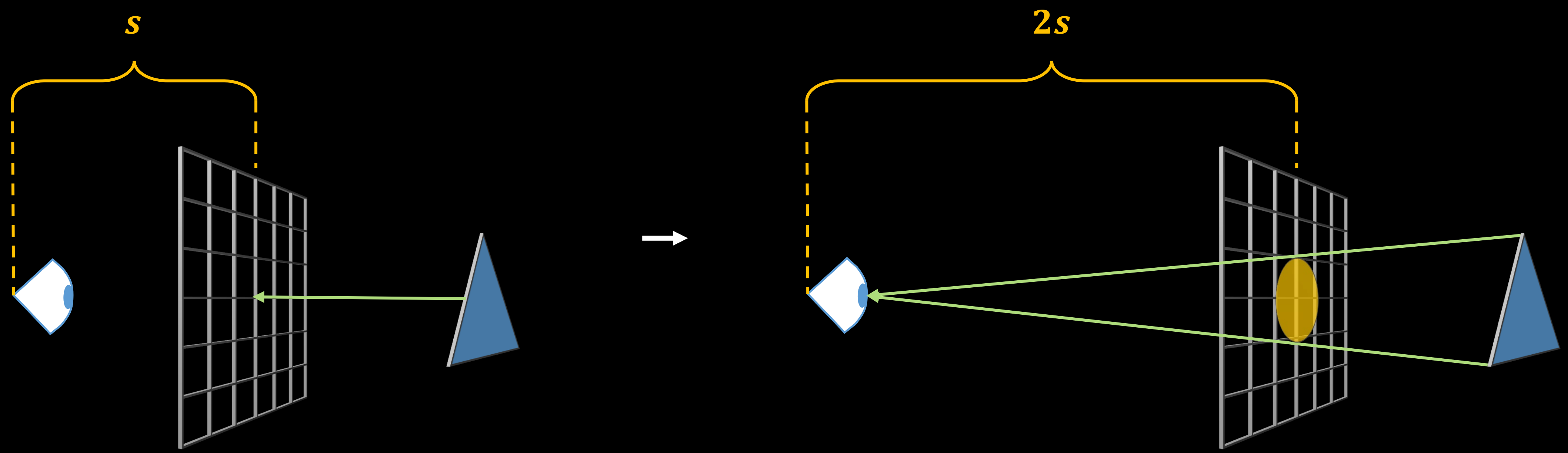
Ray Tracing: Camera Concept

- How do we translate this, so it is useful for creating computer graphics?
- s is the distance between the eye and the view plane. This distance is usually 1, but other values can be used (changing the value is like zooming in or out).



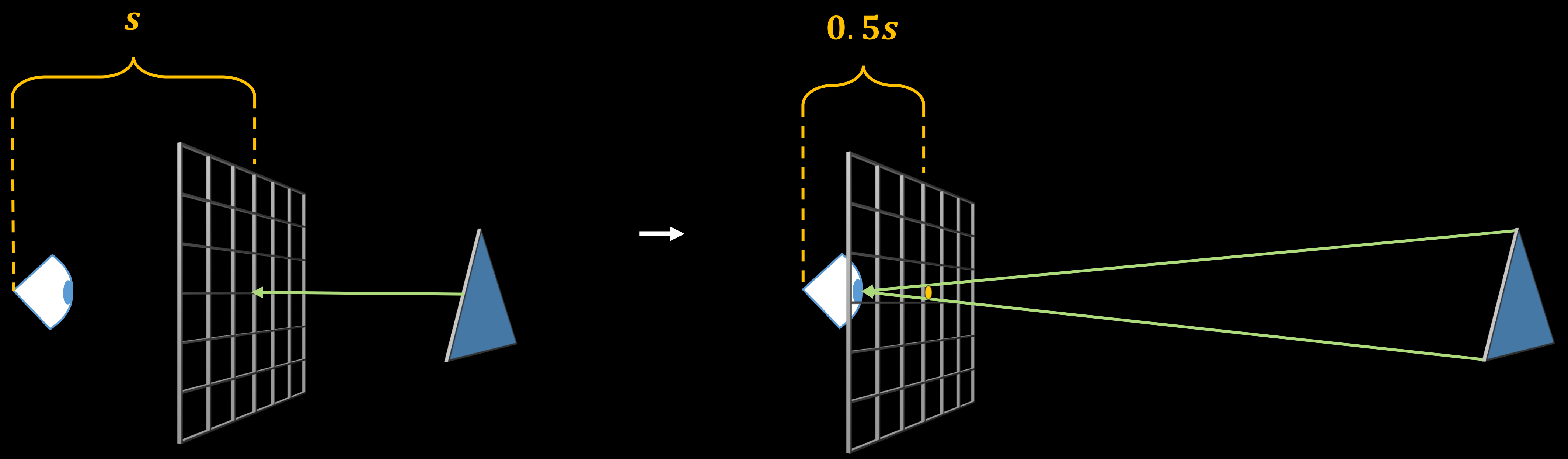
Ray Tracing: Camera Concept

- How do we translate this, so it is useful for creating computer graphics?
- s is the distance between the eye and the view plane. This is usually 1, but other values can be used (changing the value is like zooming in or out).



Ray Tracing: Camera Concept

- How do we translate this, so it is useful for creating computer graphics?
- s is the distance between the eye and the view plane. This is usually 1, but other values can be used (changing the value is like zooming in or out).

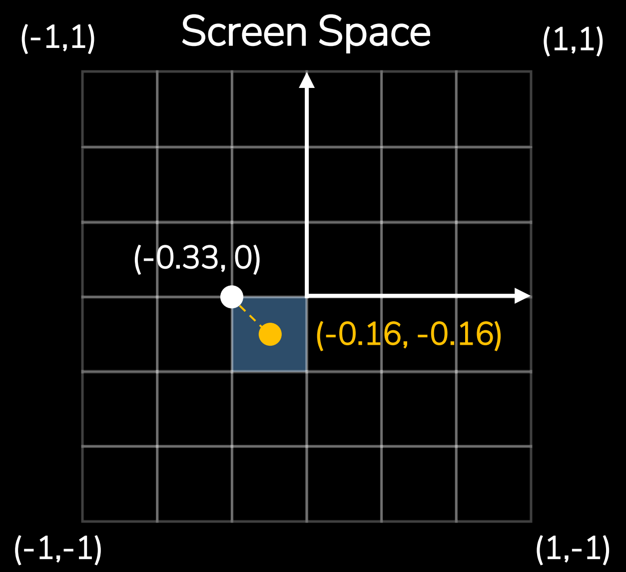


Ray Tracing: Screen Space (1SPP)

- We calculate our samples in Screen Space and transform them into World Space. But using the technique below, we assume the view plane is square!

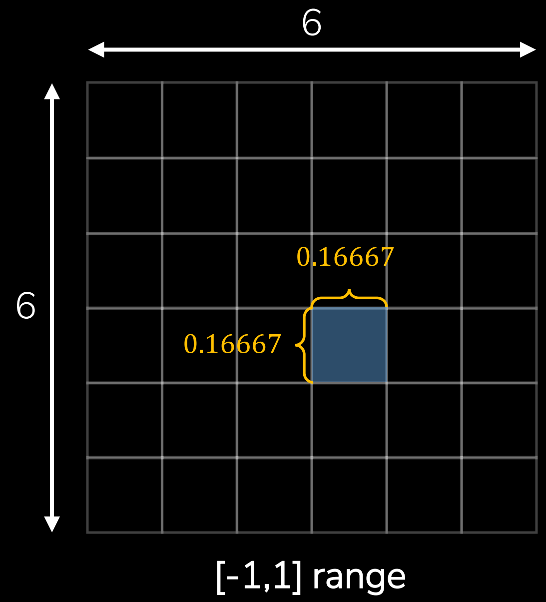
$$x_{ws} = \left(2\,\frac{(c + 0.5)}{ScreenWidth} - 1\right) \mathbf{ScreenWidth}$$

$$y_{ws} = \left(\mathbf{1} - 2\,\frac{(r + 0.5)}{ScreenHeight}\right) \quad \mathbf{ScreenHeight}$$



Ray Tracing: Screen Space (1SPP)

- We calculate our samples in Screen Space and transform them into World Space. But using the technique below, we assume the view plane is square!

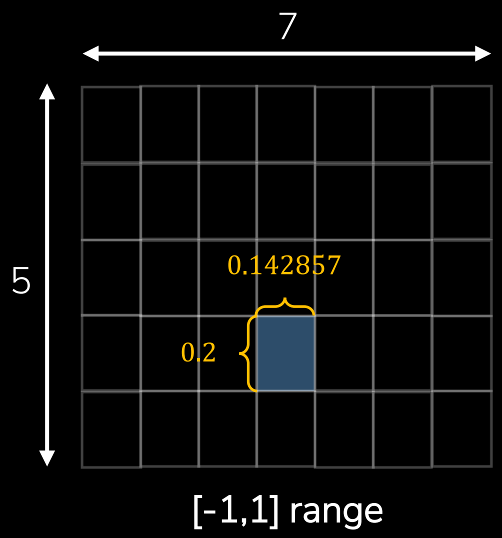


$$d_x = \frac{1}{ScreenWidth}$$

$$d_x = \frac{1}{6} = 0.16667$$

$$d_y = \frac{1}{ScreenHeight}$$

$$d_y = \frac{1}{6} = 0.16667$$



$$d_x = \frac{1}{ScreenWidth}$$

$$d_x = \frac{1}{7} = 0.142857$$

$$d_y = \frac{1}{ScreenHeight}$$

$$d_y = \frac{1}{5} = 0.2$$

- As you can see in the right example, the pixels are now <u>squashed</u>. They have a different delta value for both components. To make them square pixels again, and to make sure the image is not distorted, we must fix one of the components.

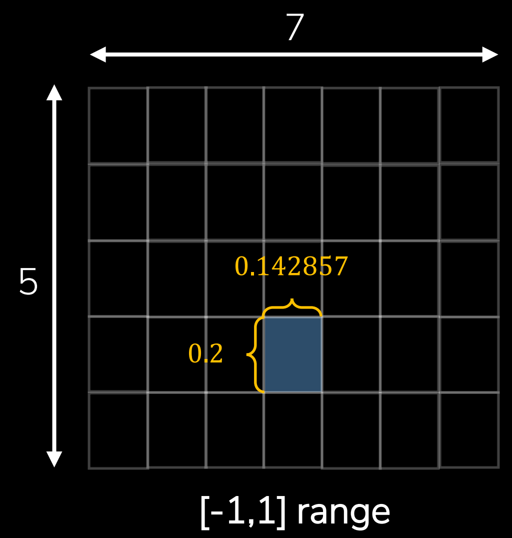
Ray Tracing: Aspect Ratio (recap)

- We usually change the ***x* component** of our pixel. We fix our component by taking into account the **aspect ratio** of the view plane. This can be done by multiplying our component with the aspect ratio.

$$\boldsymbol{AspectRatio} = \frac{\boldsymbol{ScreenWidth}}{\boldsymbol{ScreenHeight}}$$

$$x_{ss} = \left(2\,\frac{(c+0.5)}{ScreenWidth} - 1\right)\boldsymbol{AspectRatio} \longrightarrow \text{[-1.4,1.4] range}$$

$$y_{ss} = \left(1 - 2\,\frac{(r+0.5)}{ScreenHeight}\right) \longrightarrow \text{[-1,1] range}$$



$$\mathrm{d}_x = \frac{1}{ScreenWidth}$$

$$\mathrm{d}_x = \frac{1}{7} = 0.142857$$

$$AspectRatio = \frac{7}{5} = 1.4$$

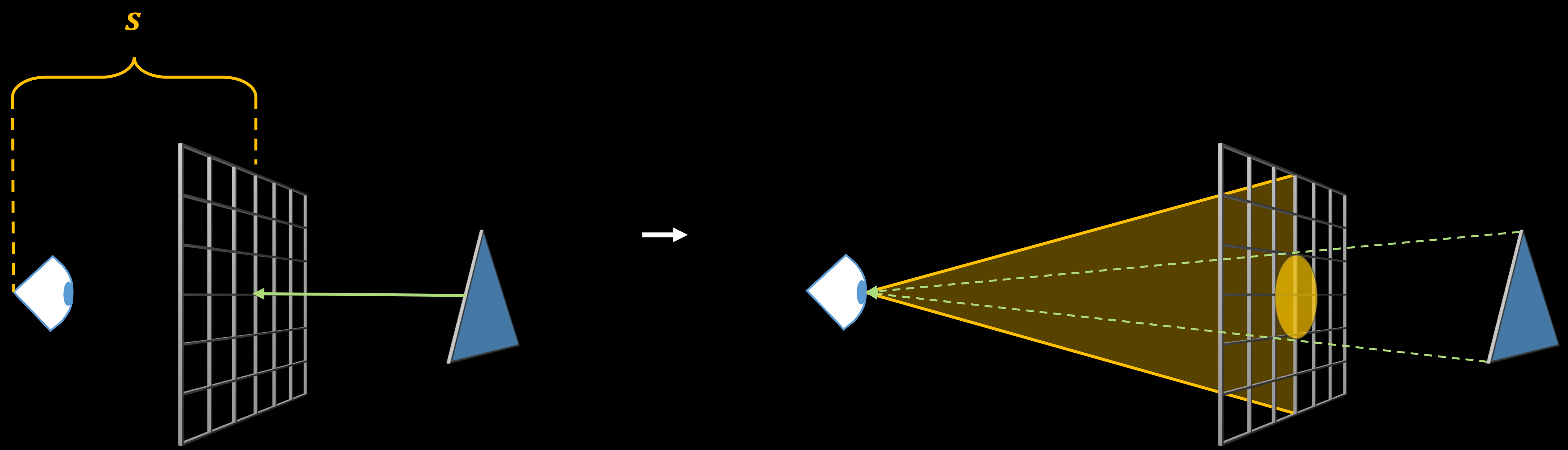
$$\mathrm{d}_x = 0.142857 * 1.4 = 0.2$$

$$\mathrm{d}_y = \frac{1}{ScreenHeight}$$

$$\mathrm{d}_y = \frac{1}{5} = 0.2$$

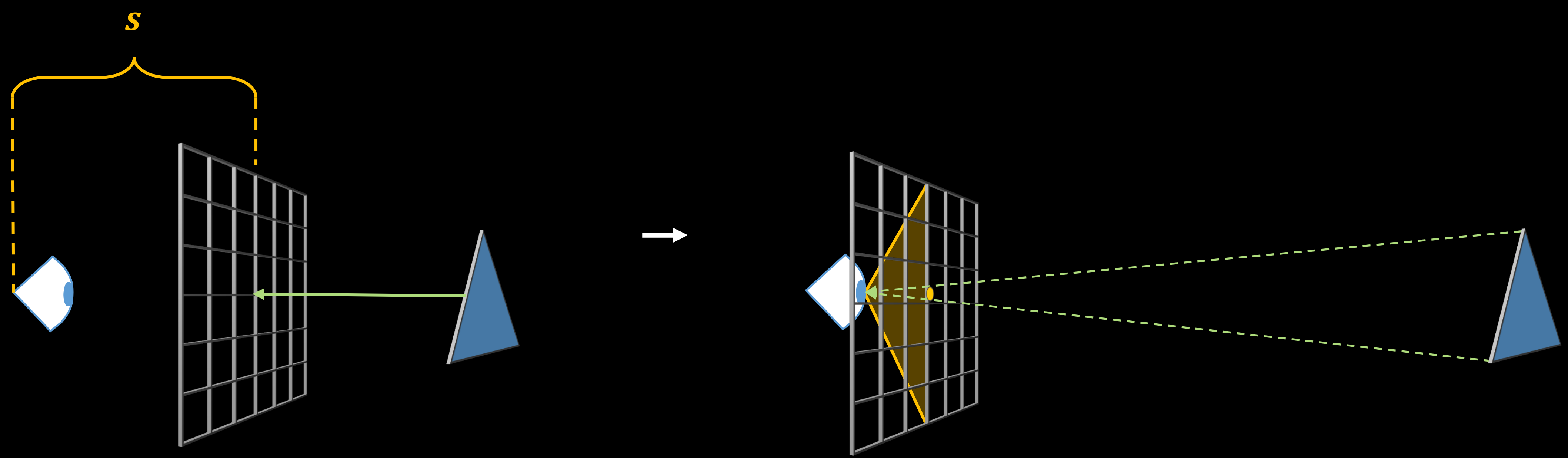
Ray Tracing: Field of View

- We talked about the distance s that can be used to zoom in or out. There is another way of doing this!



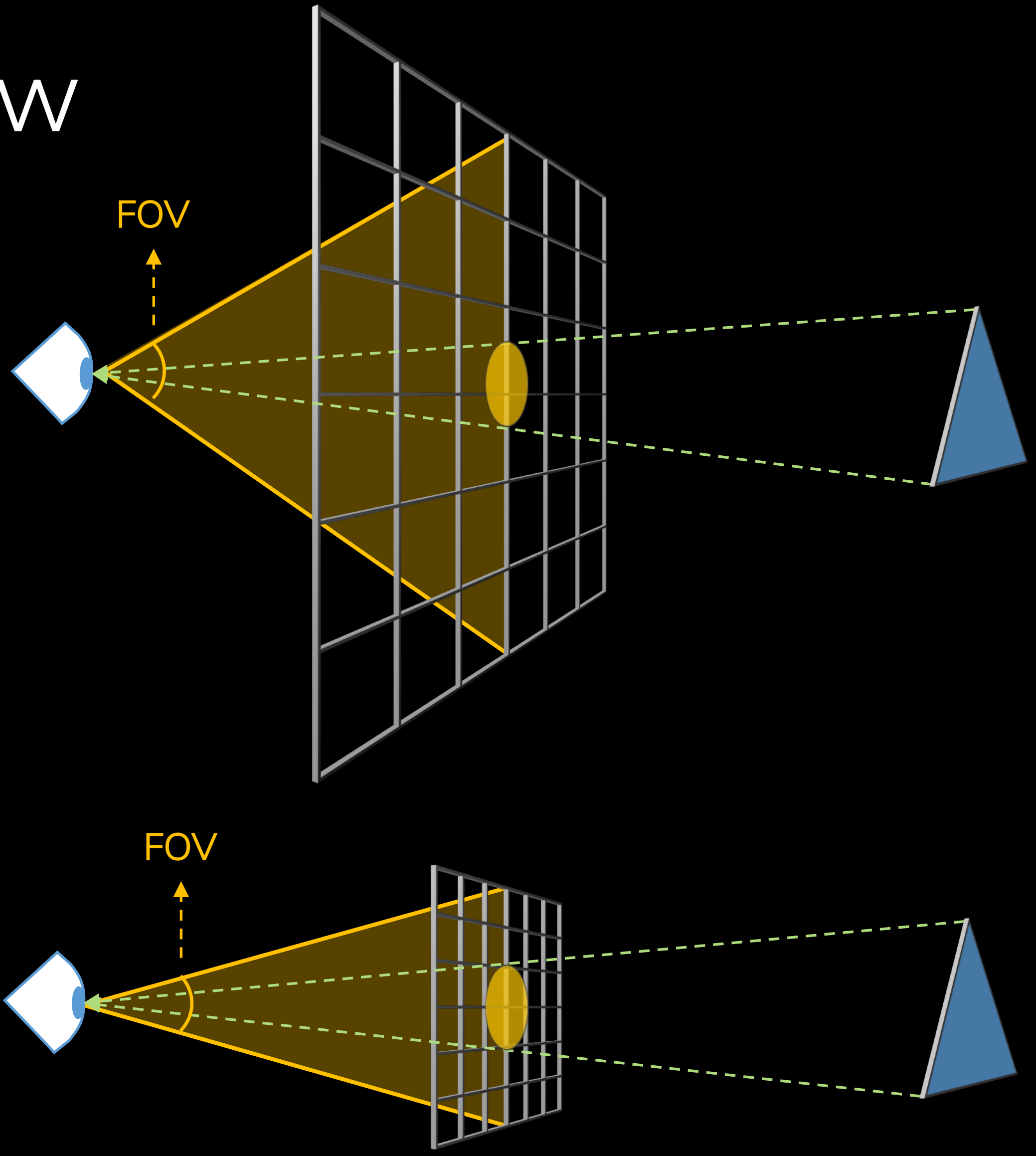
Ray Tracing: Field of View

- We talked about the distance ***s*** that can be used to zoom in or out. There is another way of doing this!



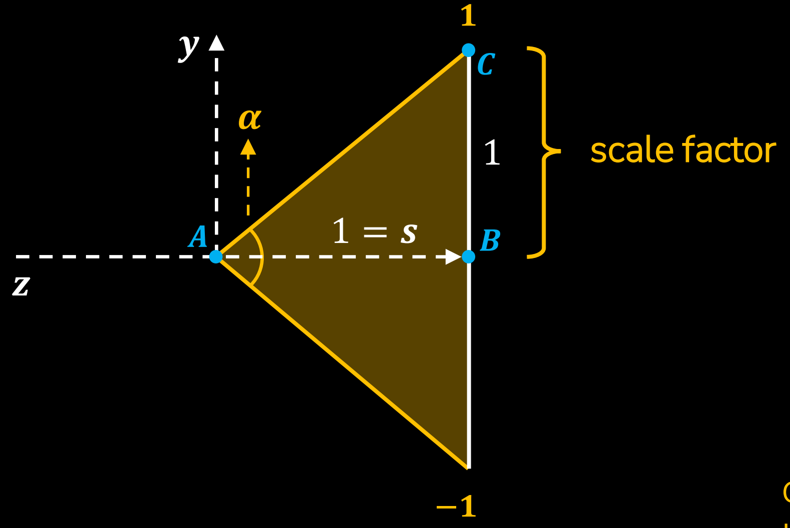
Ray Tracing: Field of View

- We talked about the distance ***s*** that can be used to zoom in or out. There is another way of doing this!
- If we change the **Field of View (FOV)**, we mimic changing the distance from our camera to the view plane. The position of the view plane doesn't change, but we **scale** the view plane, so the projection of the primitive will look smaller or bigger from our point of view!



Ray Tracing: Field of View

- How do we integrate this FOV (α) into our formula?



our scale factor,
because it's based on the [0,1] range
of our y-axis

$$\frac{\alpha}{2} = atan\left(\frac{OppositeSide}{AdjacentSide}\right) = atan\left(\frac{1}{1}\right) = \frac{\pi}{4} = 45° \longrightarrow \|BC\| = tan\left(\frac{\alpha}{2}\right)$$

Ray Tracing: Field of View

- Let’s integrate the FOV into our formula.

$$\boldsymbol{AspectRatio} = \frac{\boldsymbol{ScreenWidth}}{\boldsymbol{ScreenHeight}}$$

$$\boldsymbol{FOV} = \tan\left(\frac{\alpha}{2}\right) \text{- - -} \blacktriangleright \text{ in radians}$$

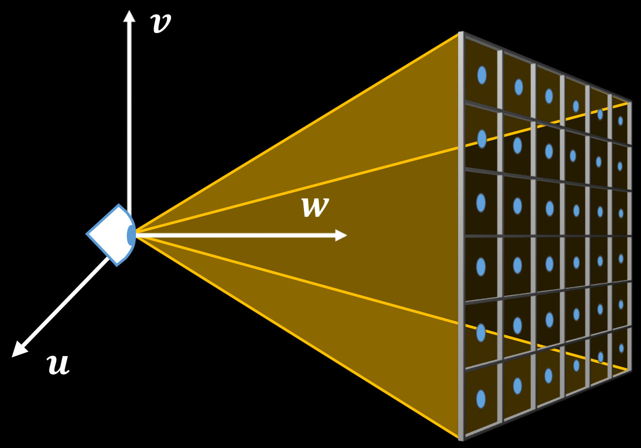
$$c_x = \left(2\,\frac{(px + 0.5)}{ScreenWidth} - 1\right)\,\boldsymbol{AspectRatio} * \boldsymbol{FOV}$$

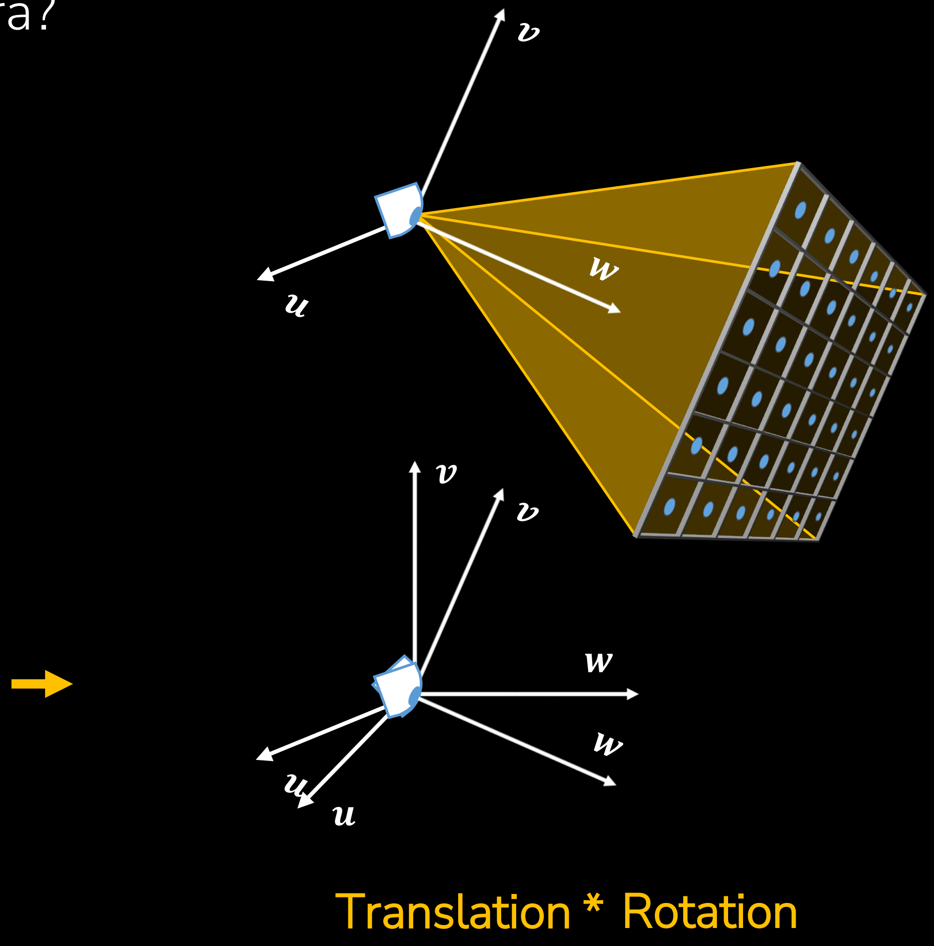
$$c_y = \left(1\ - 2\,\frac{(py + 0.5)}{ScreenHeight}\right)\,\boldsymbol{FOV}$$

- Now our x and y coordinates from our sample matches the settings of our camera, or in the other words, the camera’s view plane. This point, or sample, is now in camera space because it is expressed regarding the camera’s coordinate system.
- When we shoot rays into our scene, we do this in world space. So how do we move this sample into world space?
  - We are already in world space because we take into account the aspect ratio and the FOV. But if we would move the camera our view plane would no longer match with our camera. So how do we fix this?

Ray Tracing: World Space

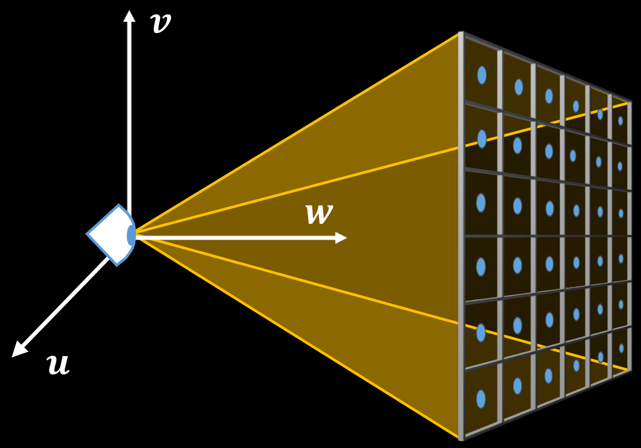
- What happens when we move a camera?

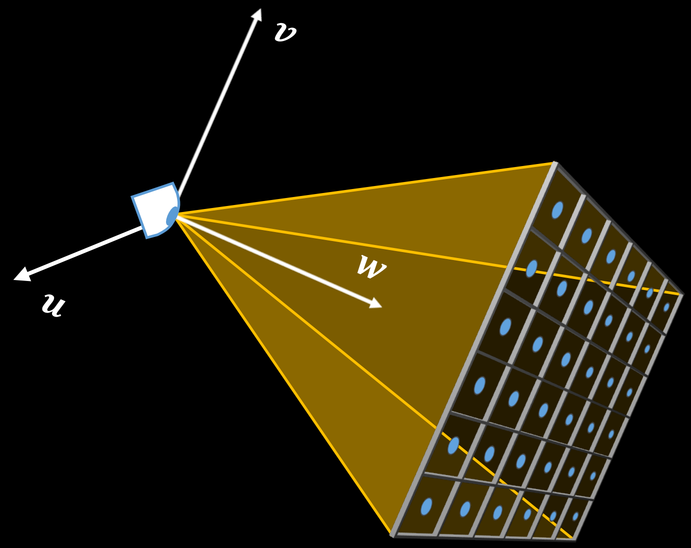




Ray Tracing: World Space

- What happens when we move a camera?
  - Camera Origin changes (Translation)
  - Camera Rotation changes (X, Y and/or Z rotations)
- What actually changes is the axis / local coordinate system of the camera. How can we represent this?





Translation * Rotation

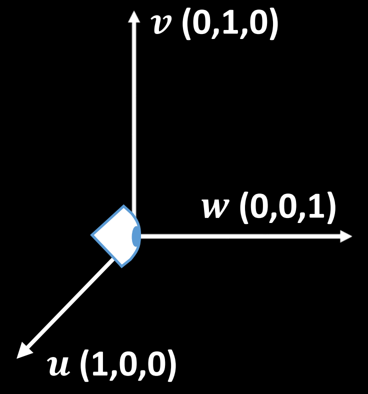
Ray Tracing: World Space

- We can use a **matrix** to store this transformation. Before we shoot a ray into the scene, we will multiple every sample with this matrix, to **map** the sample to the world space "position" of the camera. In other words, our sample will stay **relative** to the camera.
- How do we store this transformation in a matrix?

Identity Matrix

Orthonormal Basis (ONB)

$$\begin{array}{r c} \boldsymbol{u} \longrightarrow & \\ \boldsymbol{v} \longrightarrow & \\ \boldsymbol{w} \longrightarrow & \\ \text{Translation} \longrightarrow & \end{array} \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \longrightarrow [\underset{\boldsymbol{u}}{\underset{\uparrow}{Right}}\ \underset{\boldsymbol{v}}{\underset{\uparrow}{Up}}\ \underset{\boldsymbol{w}}{\underset{\uparrow}{Forward}}\ Translation]$$



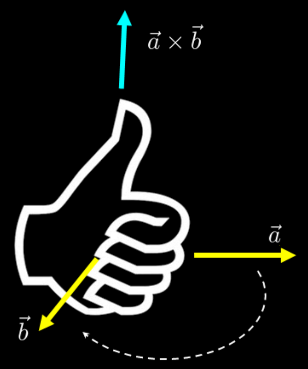
Ray Tracing: ONB

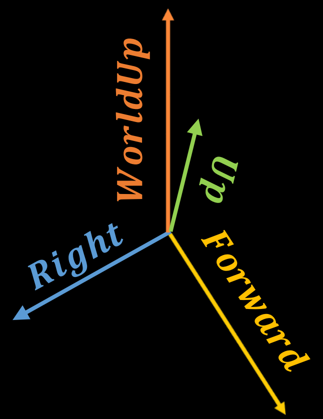
- You can easily create an ONB from two vectors by taking the cross product and normalizing the results. So, whenever the camera changes, you want to reconstruct your ONB using the following formula:

$$\boldsymbol{Right} = normalize(\boldsymbol{WorldUp} \times \boldsymbol{Forward})$$

$$\mathbf{Up} = normalize(\boldsymbol{Forward} \times \boldsymbol{Right})$$

(Left-Handed Rule)

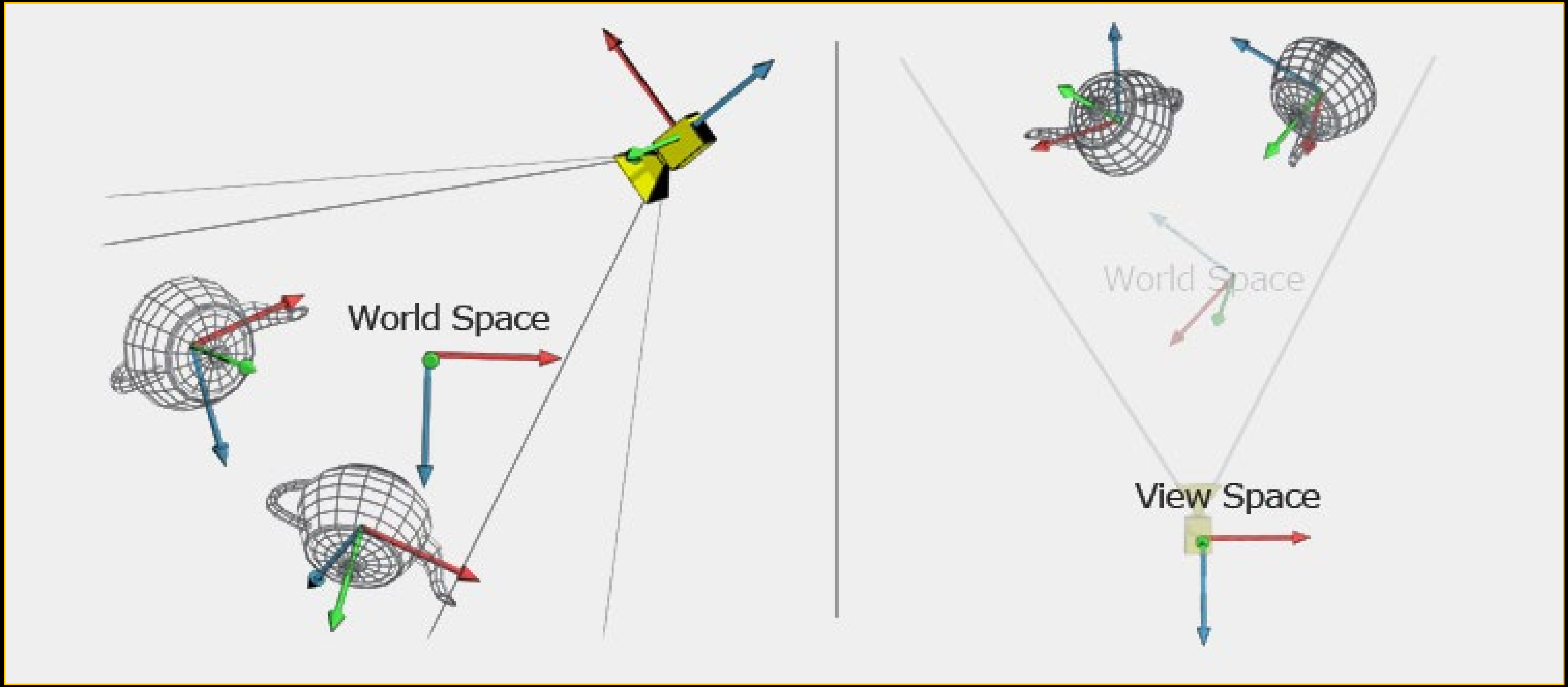




$$\rightarrow \begin{bmatrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Forward_x & Forward_y & Forward_z & 0 \\ Position_x & Position_y & Position_z & 1 \end{bmatrix}$$

- We can now use this matrix to transform our RayDirection

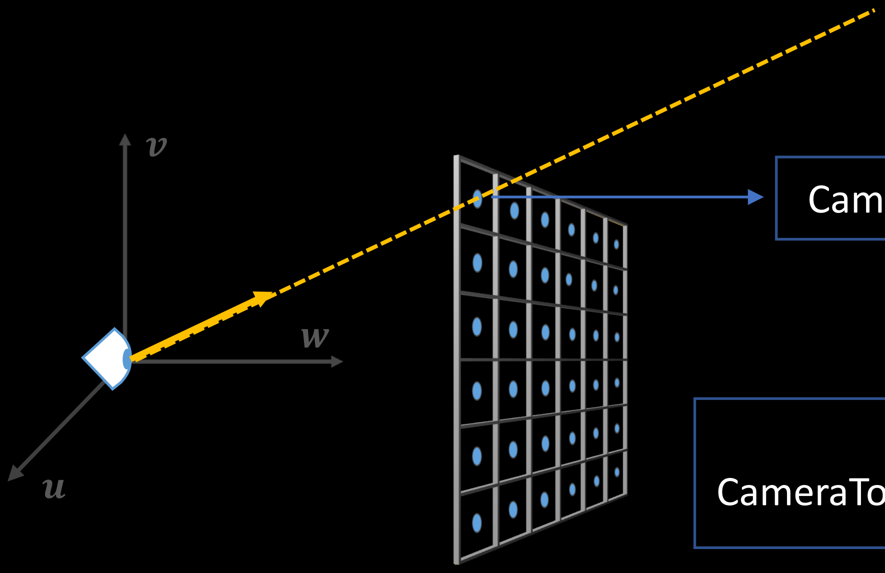
Ray Tracing: ONB



http://www.codinglabs.net/article_world_view_projection_matrix.aspx

Ray Tracing: Ray Generation

- Get the ray in the correct space (for every pixel)
  - Ray-Origin > Camera Position
  - Ray-Direction > Direction from Camera-Space to World-Space



Camera-Space Direction = (C_x , C_y , **1.0**) Why 1.0?

Transform to take camera orientation into account

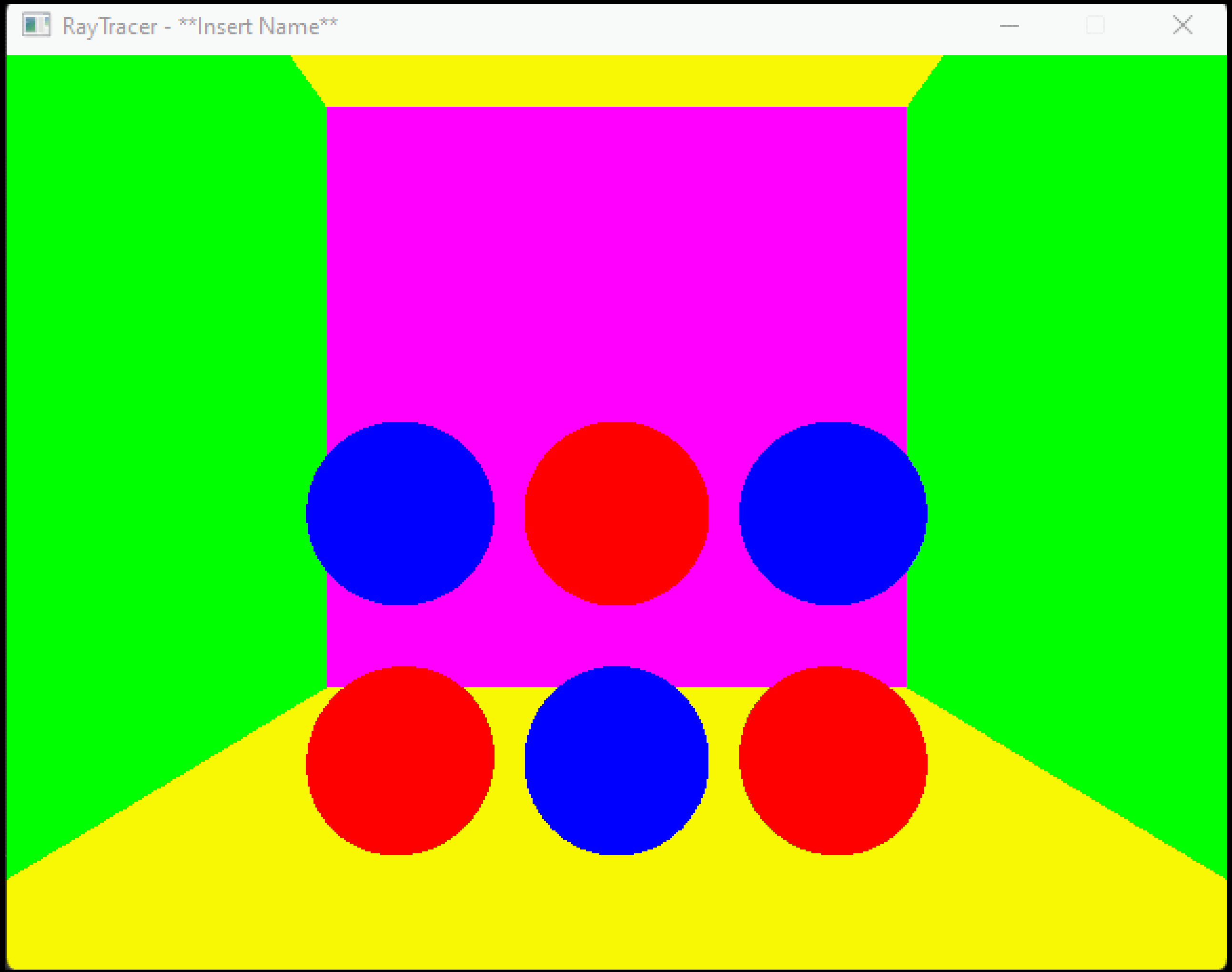
World-Space Direction =
CameraToWorld (ONB) * Camera-Space Direction

Ray Tracing: What to do (1) ?

- This week you are going to implement the following features:
  - Implement the **camera**.
  - Adjust your render loop so it uses this camera → aspect ratio, FOV and the ONB transform
    - Play with **different FOV values** and analyze the effect!
  - Capture input and move/rotate your camera real-time. Try to implement an Unreal Engine 4 editor camera.
    - For continuous events don't use the PollEvent in main.cpp because it's dependent on the repeat rate of your OS. Instead use:
      - SDL_GetKeyboardState(…) and SDL_GetRelativeMouseState(…)
    - Provide extra functions so you can change the location and rotation (pitch & yaw) of the camera.
  - Generate Rays (taking CameraToWorld Matrix (ONB) into account!)
  - Perform Hit-Tests (same as previous lab)

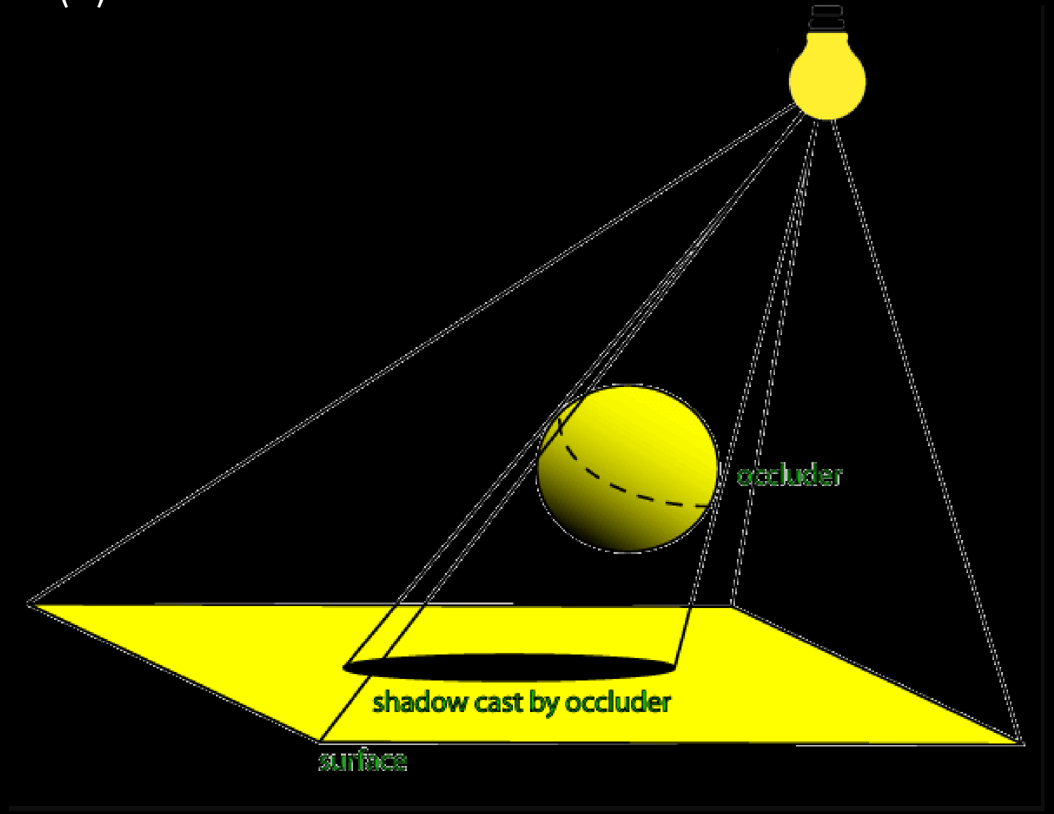
Control	Action
Perspective	
LMB + Drag	Moves the camera forward and backward and rotates left and right.
RMB + Drag	Rotates the viewport camera.
LMB + RMB + Drag	Moves up and down.

Ray Tracing: Result



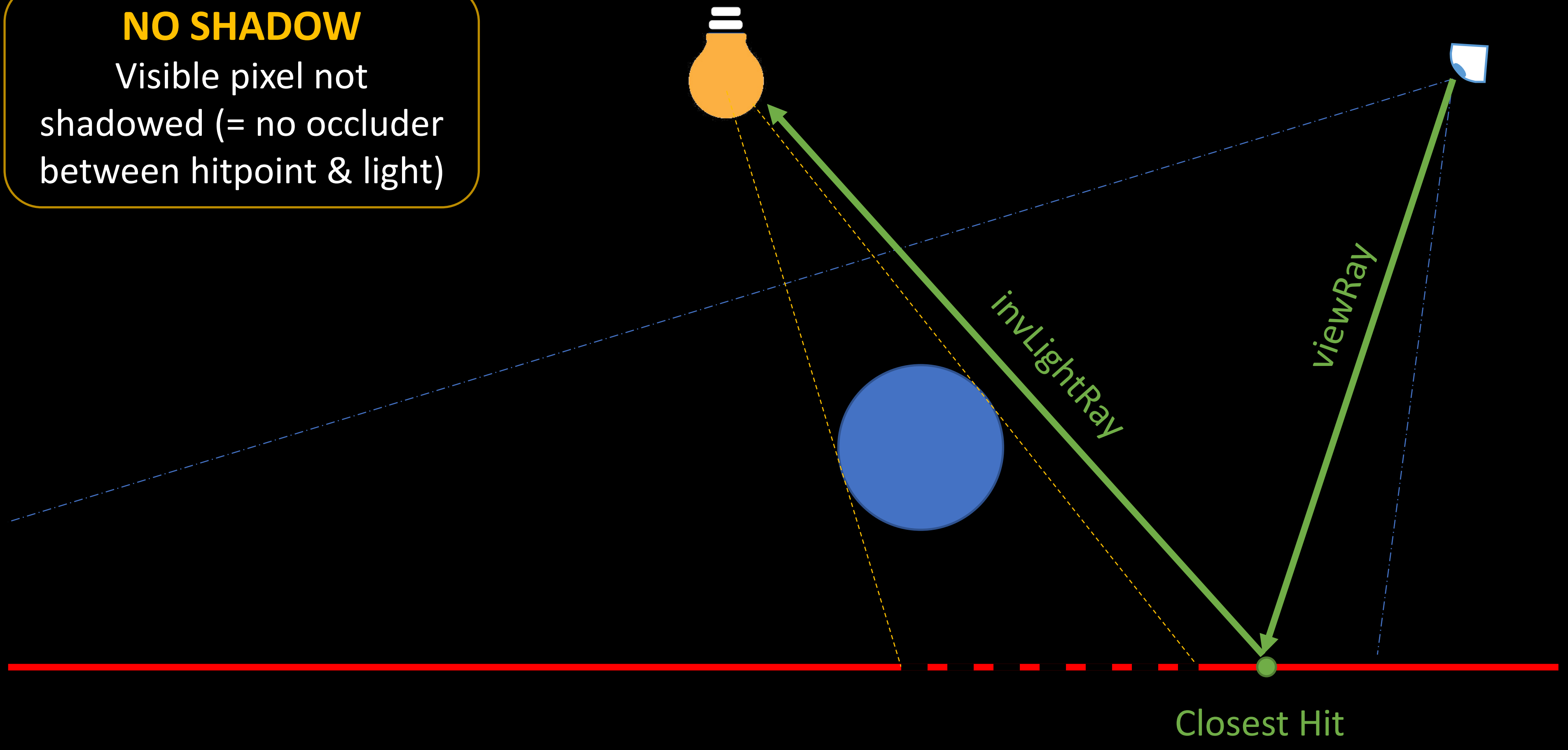
Ray Tracing: Hard Shadows

- Implementing (Hard) Shadows is very simple, you almost get it for free!
- TODO: Check if some object (=occluder) is blocking the path between a (visible) hitpoint and the light source(s)



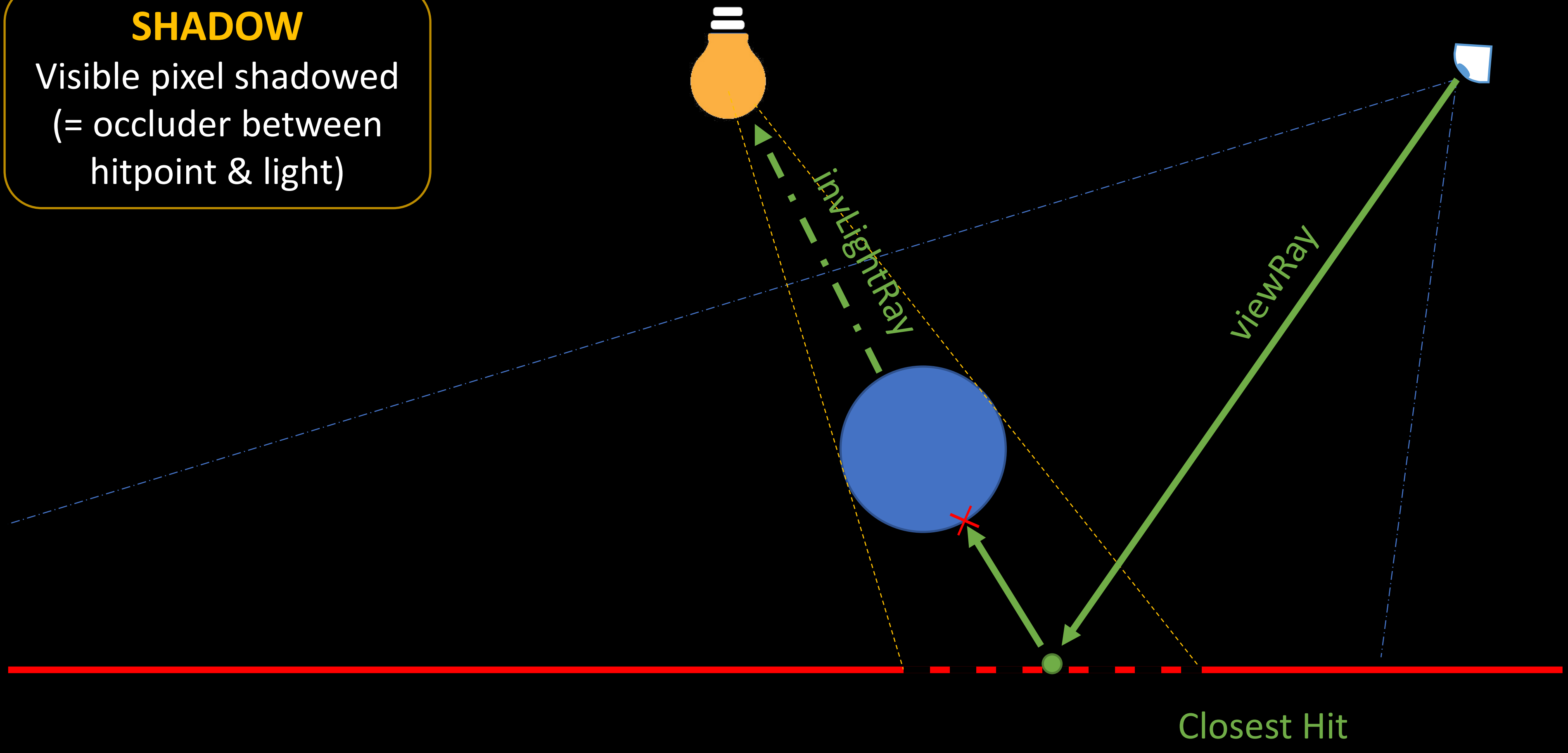
Ray Tracing: Hard Shadows

NO SHADOW
Visible pixel not shadowed (= no occluder between hitpoint & light)



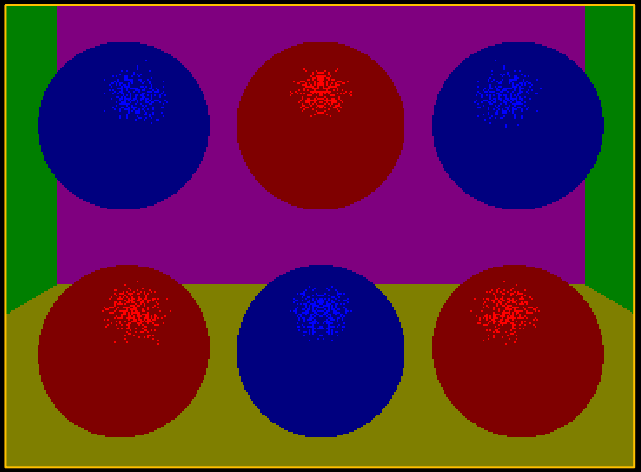
Ray Tracing: Hard Shadows

SHADOW
Visible pixel shadowed
(= occluder between
hitpoint & light)

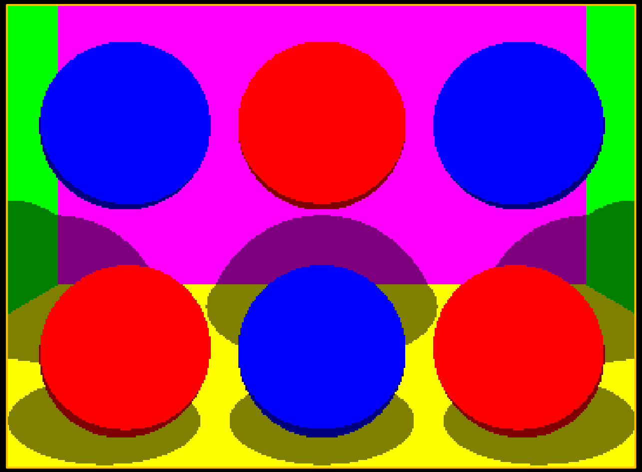


Ray Tracing: Hard Shadows

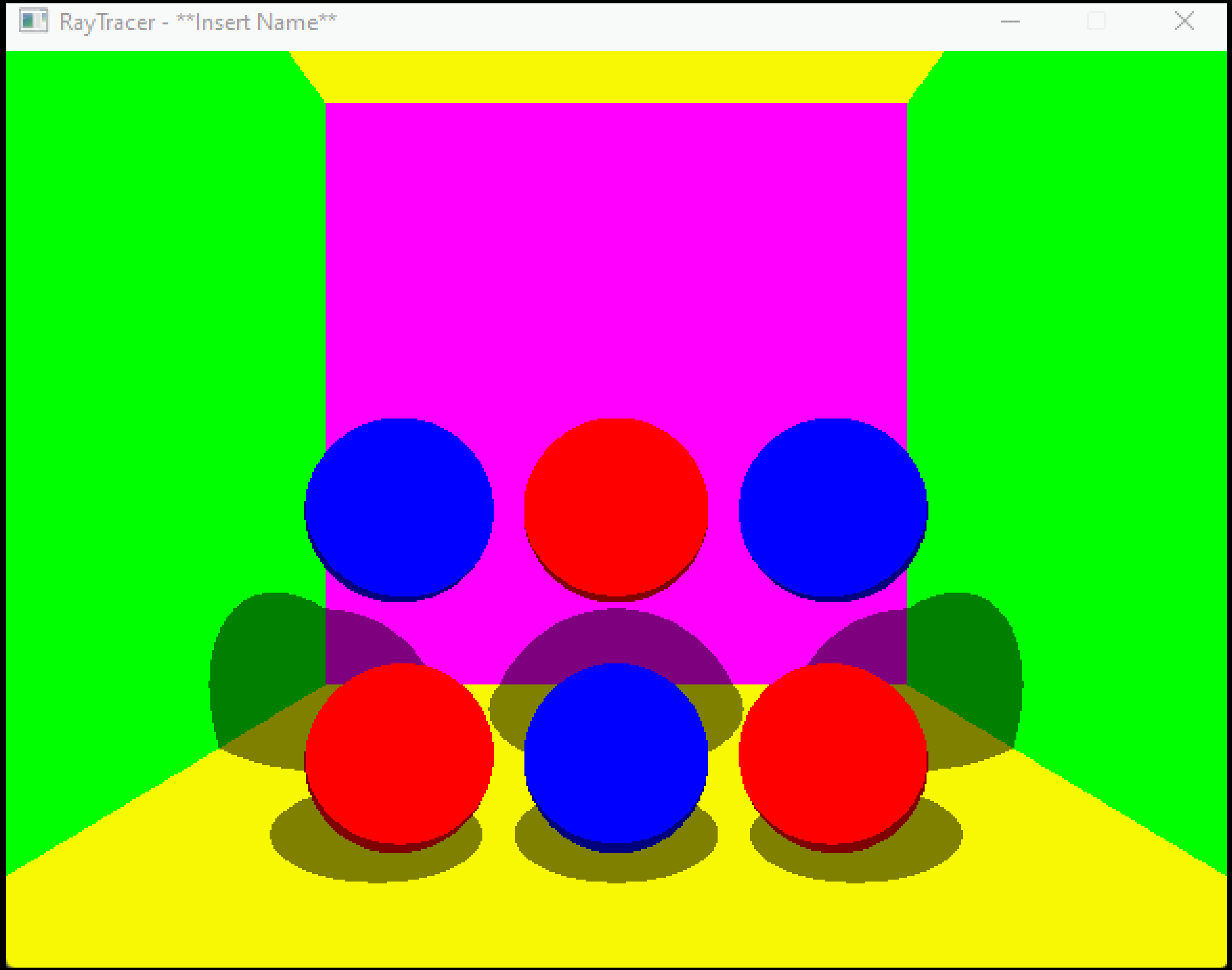
- If your ray starts at the position we just hit, it's possible you get something called **self-intersection**. In other words, your DoesHit(…) function will say it hits a primitive. The primitive that has the actual point. How can we solve this?
- We just **move** our start position upwards based on the normal of the surface, or we change the **tMin** of our ray to a small value.
- Finally, you don't need to check to infinity (**tMax**). You can use the **length** of the light vector (you need to normalize it anyway).



→

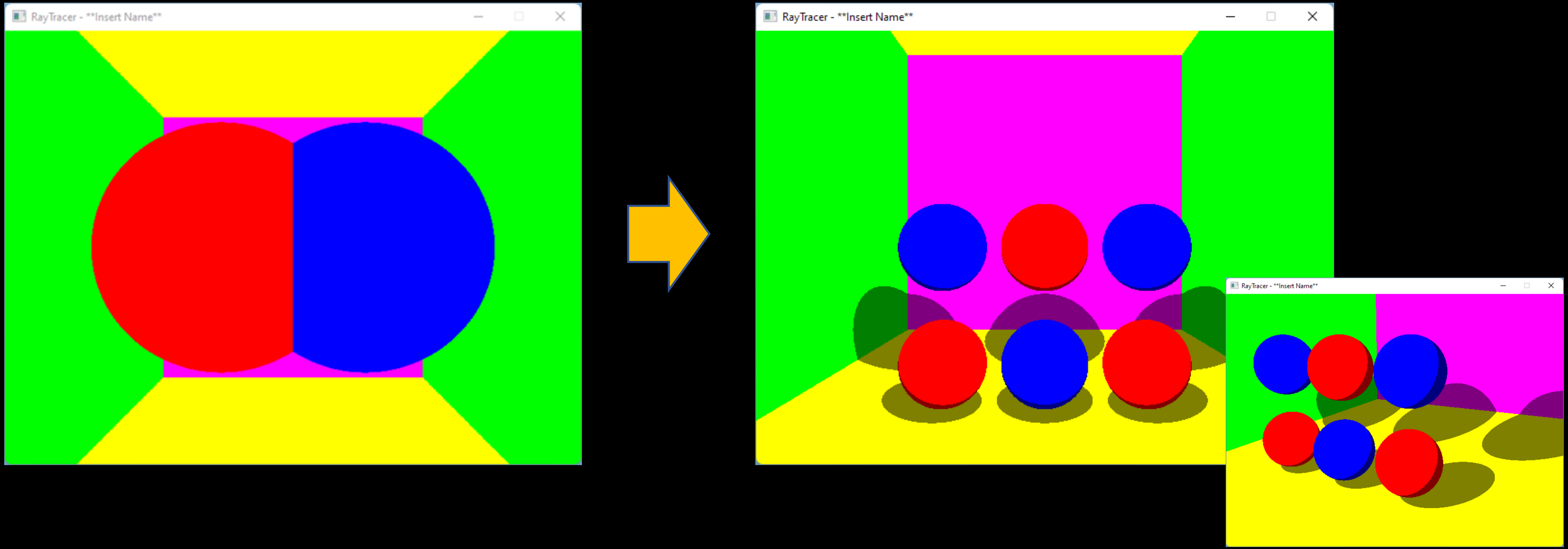


Ray Tracing: Hard Shadows



W2 Lab - Objective

- Objective
  1. Implement Camera (Movement/Rotation, RayDirection Transform, FOV)
  2. Implement Hard Shadows



W2 Lab – Objective [Summary]

- Create & Initialize Scene_W2
- Implement Matrix CreateRotation + variants
- Implement Matrix CreateScale
- Implement Camera::CalculateCameraToWorld
- Implement Camera::Update
- Update Renderer::Render >> CAMERA
  - Cache cameraToWorld matrix
  - Cache aspectRatio (last week)
  - Calculate & cache fov (radians vs degrees!)
  - Update cx/cy calculation (order of operations!)
  - Transform rayDirection (Vector!)

- Update Renderer::Render >> SHADOWS
  - Iterate Lights > Check for occluder between hitpoint and light
  - If blocked >> final color is darkened

W1 Lab – Todo (1)

Create and Intialize a new Scene > Scene_W2

- Use the implementation of ‘Scene_W1’ as a guide to implement a new scene ‘Scene_W2’
- Also make sure that Scene_W2 is the active scene!

Main.cpp

```cpp
//const auto pScene = new Scene_W1();
const auto pScene = new Scene_W2();
pScene->Initialize();
```

- The scene contains
  - 5 planes
  - 6 spheres
  - 1 point light
    - Only the light’s origin is important for this lab
  - See Initialization Details (screenshot)

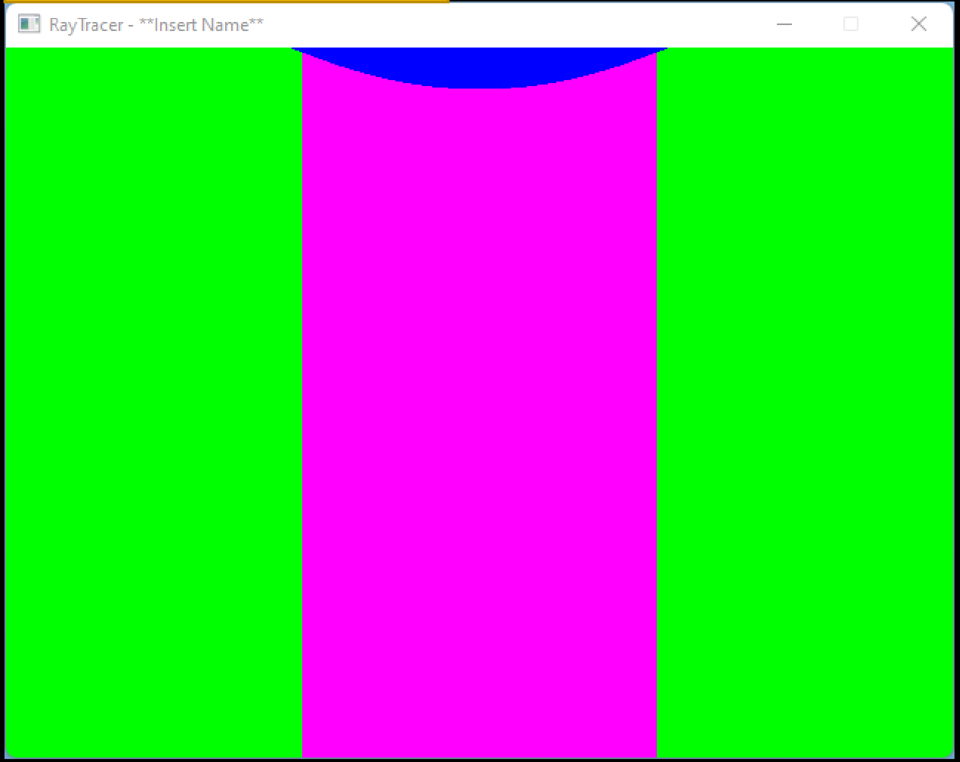
Scene_W2 Initialization Details

```cpp
void Scene_W2::Initialize()
{
    m_Camera.origin = { _x:0.f, _y:3.f, _z:-9.f };
    m_Camera.fovAngle = 45.f;

    //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 });
    const unsigned char matId_Solid_Green = AddMaterial(new Material_SolidColor{ colors::Green });
    const unsigned char matId_Solid_Magenta = AddMaterial(new Material_SolidColor{ colors::Magenta });

    //Plane
    AddPlane(origin: { _x:-5.f, _y:0.f, _z:0.f }, normal: { _x:1.f, _y:0.f, _z:0.f }, matId_Solid_Green);
    AddPlane(origin: { _x:5.f, _y:0.f, _z:0.f }, normal: { _x:-1.f, _y:0.f, _z:0.f }, matId_Solid_Green);
    AddPlane(origin: { _x:0.f, _y:0.f, _z:0.f }, normal: { _x:0.f, _y:1.f, _z:0.f }, matId_Solid_Yellow);
    AddPlane(origin: { _x:0.f, _y:10.f, _z:0.f }, normal: { _x:0.f, _y:-1.f, _z:0.f }, matId_Solid_Yellow);
    AddPlane(origin: { _x:0.f, _y:0.f, _z:10.f }, normal: { _x:0.f, _y:0.f, _z:-1.f }, matId_Solid_Magenta);

    //Spheres
    AddSphere(origin: { _x:-1.75f, _y:1.f, _z:0.f }, radius:.75f, matId_Solid_Red);
    AddSphere(origin: { _x:0.f, _y:1.f, _z:0.f }, radius:.75f, matId_Solid_Blue);
    AddSphere(origin: { _x:1.75f, _y:1.f, _z:0.f }, radius:.75f, matId_Solid_Red);
    AddSphere(origin: { _x:-1.75f, _y:3.f, _z:0.f }, radius:.75f, matId_Solid_Blue);
    AddSphere(origin: { _x:0.f, _y:3.f, _z:0.f }, radius:.75f, matId_Solid_Red);
    AddSphere(origin: { _x:1.75f, _y:3.f, _z:0.f }, radius:.75f, matId_Solid_Blue);

    //Light
    AddPointLight(origin: { _x:0.f, _y:5.f, _z:-5.f }, intensity:70.f, colors::White);
}
```

W1 Lab – Todo (1)

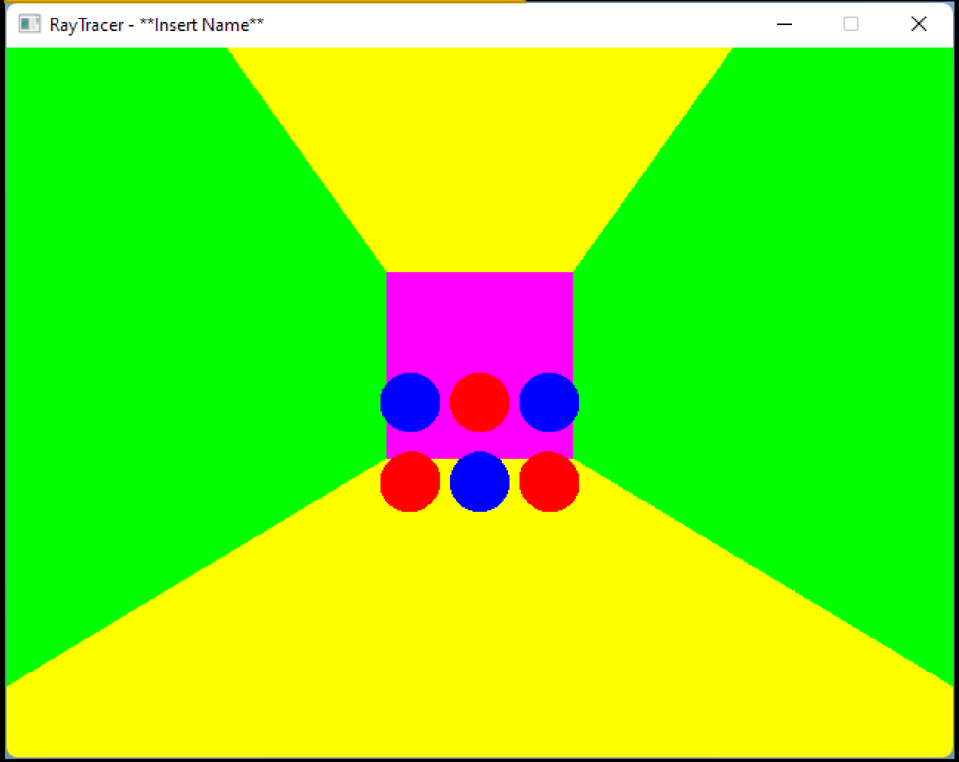
With the code from last week, the scene should look like this (double check your viewRay origin!):

viewRay origin = {0, 0, 0}




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

viewRay origin = camera origin

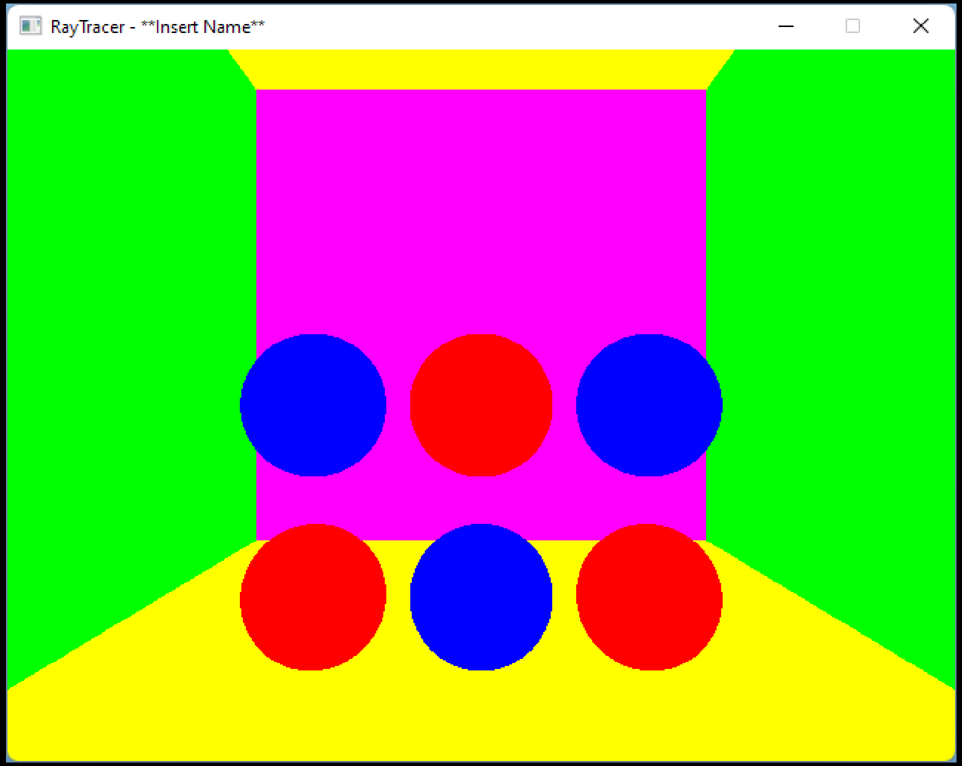



```cpp
//Ray we are casting from the camera towards each pixel
Ray viewRay{ camera.origin, rayDirection };
```

W1 Lab – Todo (2) [Verify]

Take the camera's FOV into account
(Should only be calculated once per frame – this could be further optimized by calculating the 'fov' during camera construction, and only update it if the fovAngle changes)

Also, use brackets in your calculations to ensure a correct order of calculations



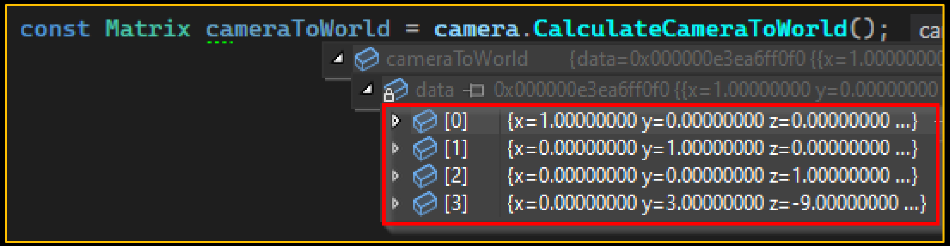
W1 Lab – Todo (3)

RayDirection Transform

1. Implement Camera::CalculateCameraToWorld
   - This function should return the Camera ONB matrix
   - Calculate the right & up vector using the forward camera vector
   - Combine to a matrix (also include origin) and return

2. Transform your RayDirection with this matrix (Renderer::Render) – It's a vector, not a point!

Because the default forward vector of the camera is equal to the World Z+ axis, the Camera::CalculateCameraToWorld function should return a (semi) Identity Matrix, with the translation component equal to the origin of the camera



W1 Lab – Todo (3) [Verify]

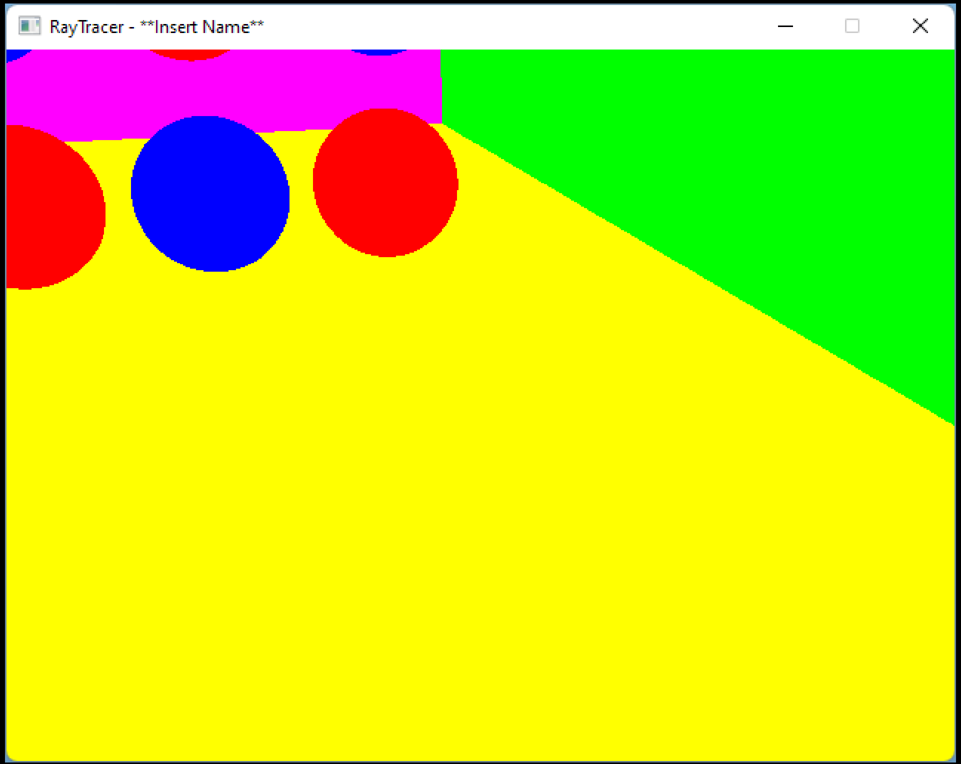
Verify your ONB calculations

We can manually rotate the camera by altering the default forward vector, temporarily change your camera forward to the coordinates defined below and verify your result.

Camera.h

```
Vector3 forward{ _x: 0.266f, _y: -0.453f, _z: 0.860f};
Vector3 up{Vector3::UnitY};
Vector3 right{Vector3::UnitX};
```

results in



W1 Lab – Todo (4)

Camera Movement/Rotation

1. Movement = Transforming the camera's origin (WASD || LMB + MouseY || LMB + RMB + MouseY)
2. Rotation = Transforming the camera's forward vector (LMB + MouseX || LMR + MouseX/Y)

Implement the required logic inside the Camera::Update function – make sure to make your implementation framerate independant (= use deltaTime)

- Use some constants to control the Movement & Rotation speed
- Focus on the Keyboard Movement first
  - WASD > Move Forward/Backward/Left/Right
  - Checking for KeyBoard Input > pKeyboardState[SDL_SCANCODE_...] (1=pressed, 0=not pressed)
    - Check SDL Documentation for more info

W1 Lab – Todo (4)

Next, Mouse Movement/Rotation (see documentation for SDL_GetRelativeMouseState)

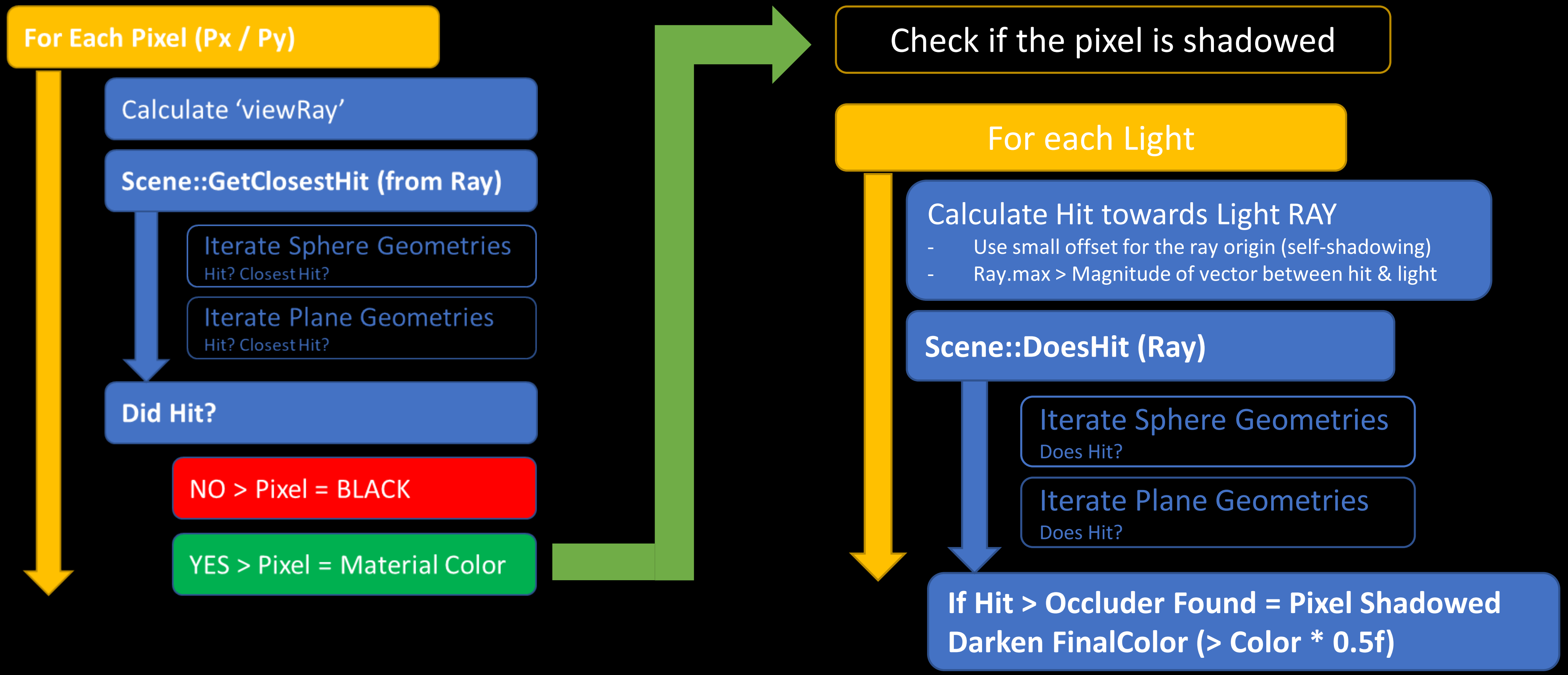
1. Implement the Matrix::CreateRotation(X/YZ) functions
2. Based on the mouse button states and mouse movement, alter the total Pitch/Yaw angle
3. At the end of the update function, create a Rotation Matrix (using the totalPitch & totalYaw)
4. Use this Rotation Matrix to calculate the current forward vector for the camera

```
forward = finalRotation.TransformVector( v: Vector3::UnitZ);
forward.Normalize();
```

Verify your implementation with the demo application found on Leho!

W1 Lab – Todo (5)

Implement (Hard) Shadows,
the flow of your renderer should look like this:



W1 Lab – Todo (5)

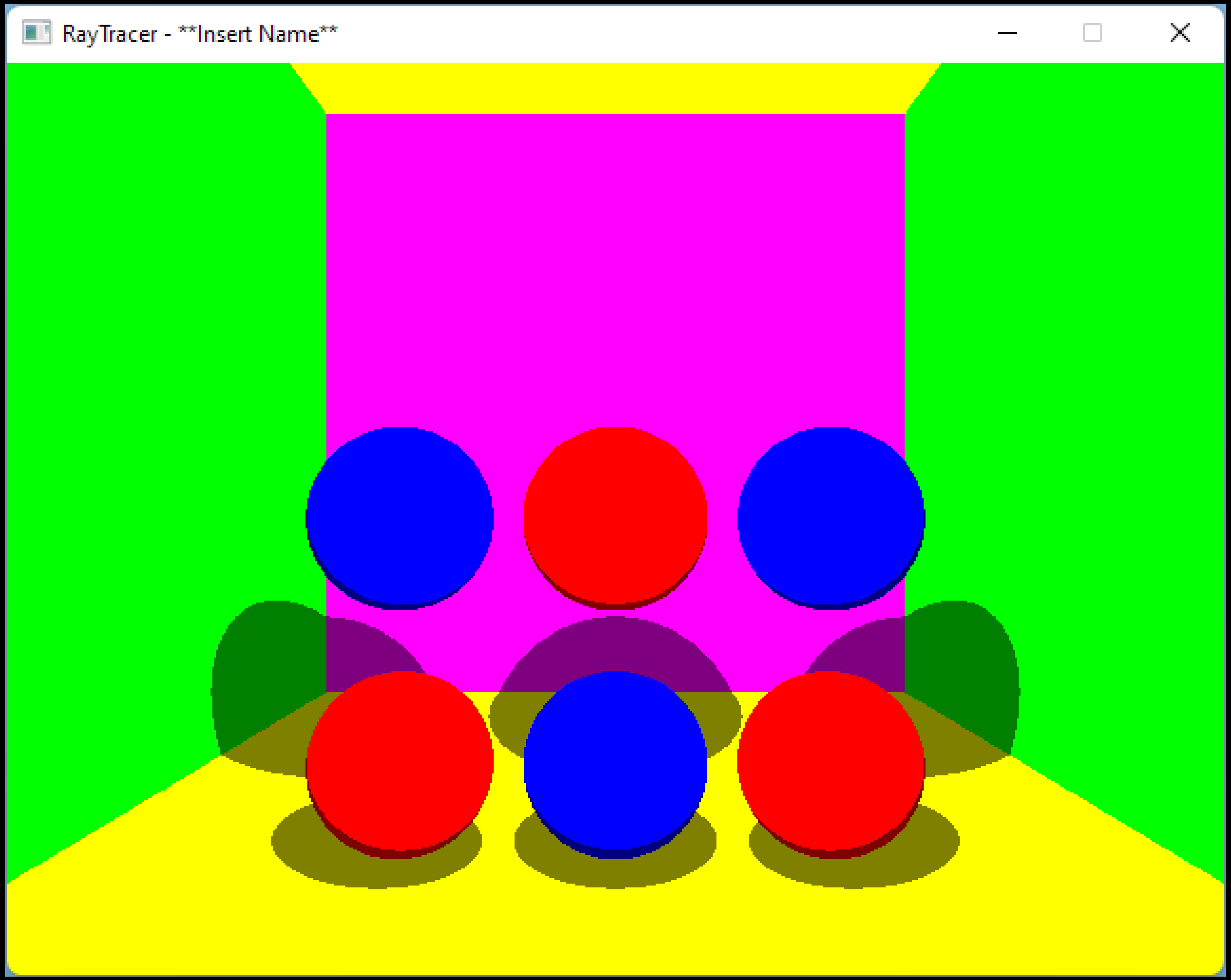
In order to implement (Hard) Shadows, you'll have to

1. Implement the Scene::DoesHit(…) function, this function should return true on the first hit for the given ray, otherwise false. (No need to check for the closest hit, or filling in the HitRecord…)
2. Implement the LightUtils::GetDirectionToLight(…), this function should return a unnormalized vector going from the origin to the light's origin.
   The implementation depends on the light type because a directional light does not have an origin, and the magnitude of this direction would be equal to FLT_MAX.
   1. Because the returned vector is Unnormalized, you can perform the normalization call inside your shadowing logic (Renderer::Render) and automatically capture the magnitude (distance between hit & light)

Light Ray

Origin > Offset Point (offset along the normal of the original hitpoint)
Direction > Hit to Light Direction (Normalized!)
Min > 0.0001f
Max > Distance between hit & light

W1 Lab – Todo (5) [Verify]



GOOD LUCK!