# Compsci.373 Tutorial 6

COMPUTER GRAPHICS II

# Today’s outline

- Getting ready for the first programming assignment
- A detailed look at the Ray Casting process
- The mathematics behind the Ray Casting algorithm

# Your first programming assignment

Your first programming assignment will include questions on Ray Casting material you have seen up to now in lectures, but will also include some questions on vector and matrix operations.

# Your first programming assignment

The assignment will be entirely in C, but will involve little more than you are already used to with Java.

There is a C/C++ tutorial on the CS website, and I encourage you all to have a look at it (You can ignore the C++ section):

https://www.cs.auckland.ac.nz/courses/compsci373s1c/BurkhardsLectures/IntroductionCandC++.pdf

# Getting prepared for the assignment

A skeleton version of the Ray Caster source code will become available by the end of the week. There will be a number of incomplete functions which we leave for you to complete.

Unlike previous years, the assignment will be entirely assessed online. You will be given a limited amount of time, and a set of coding questions to complete during that time.

You will be asked to complete various functions of the ray caster, so it is a good idea to work on the skeleton code before you attempt the assignment!

# Vector operations in code

Because ray casting is predominantly a series of vector operations, we will need a set of functions to implement these operations.

To accommodate this, we are providing the skeleton code for a very simple linear algebra library which supports most of the basic vector and matrix operations you have seen up to now.

# Vector operations in code

It provides two types:

A 3 dimensional vector and a 3x3 matrix.

And 10 linear algebra operations: Vector addition, vector subtraction, vector scaling, vector normalizing, vector projection, vector magnitude, vector dot product, matrix-vector multiplication, and matrix-matrix multiplication.

# The Vector3f type

```
typedef struct {
    float x;
    float y;
    float z;
} Vector3f;
```

A struct is a bit like a mix between classes and primitive variables in java. It has a series of fields that you can set on it, but you don’t have to instantiate it before you use it.

In this case, the Vector3f has three fields. One for its x, y, and z components. You can access the components using either v.x, v.y, v.z or v->x, v->y, v->z. Which one you use depends on whether your vector (v in this case) is a pointer or not.

# The Matrix3f type

```
typedef float Matrix3f[3][3];
```

The Matrix3f type is an alias for a 3x3 2-dimensional array. 2-dimensional arrays work the same as in Java: mymat[0][2] would be the element at row 0, column 2 of the matrix ‘mymat’.

# Implementing a vector addition function

```
Vector3f homogeneousVectorAdd(Vector3f* vec1, Vector3f* vec2) {
    Vector3f retVec;
    retVec.x = vec1->x + vec2->x;
    retVec.y = vec1->y + vec2->y;
    retVec.z = 1;
    return retVec;
}
```

This function simply creates a new Vector3f, and sets each of its components to be the addition of the corresponding components in the two function parameter vectors. It is important to note that the z component is set to 1. We’re adding two homogeneous vectors, which means that the z component must always be 1!

# Implementing a function to get a vector's magnitude

```
float homogeneousVectorMagnitude(Vector3f* vec) {
    return sqrt(vec->x * vec->x + vec->y * vec->y);
}
```

This function takes a vector and returns the square root of all of the vectors components squared. Again, because this is a homogeneous vector, we do not consider the z component.

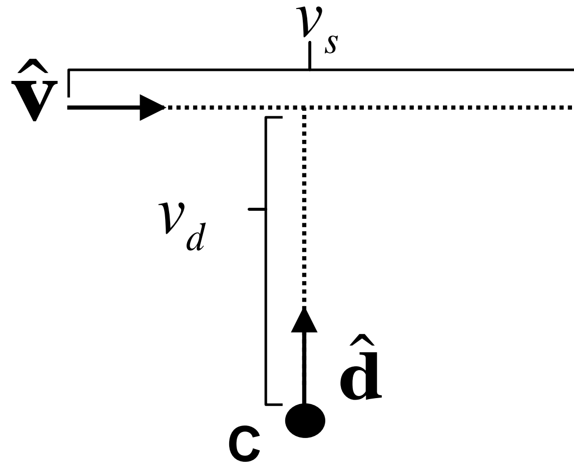
# Lets look at some code!

# 2D Ray Casting

2D Ray Casting is an identical concept to the 3D Ray Tracing you’ve seen earlier, only far simpler!

# The Ray Casting viewplane

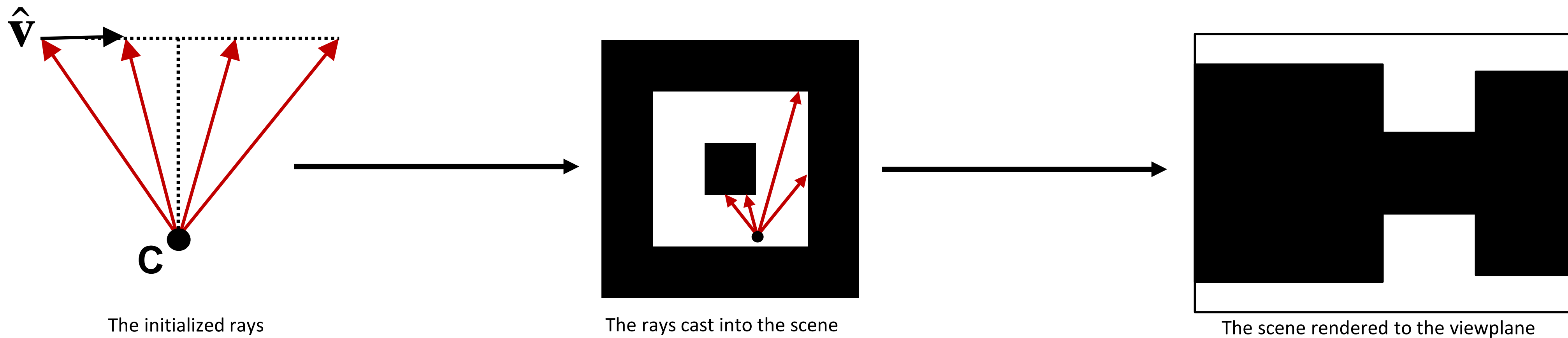
We still have a viewplane, but because we're dealing with a lower dimension to 3D Ray Tracing, it's just a vector.



# The Ray Casting viewplane

It helps to think of the viewplane as being a window to the world. Whatever the camera can see through it is what gets drawn.

In order to find out what the camera can see, we draw rays starting from the camera and crossing through the viewplane at regular intervals. These rays extend until they hit something, and whatever they hit is what gets drawn using vertical pixel columns.



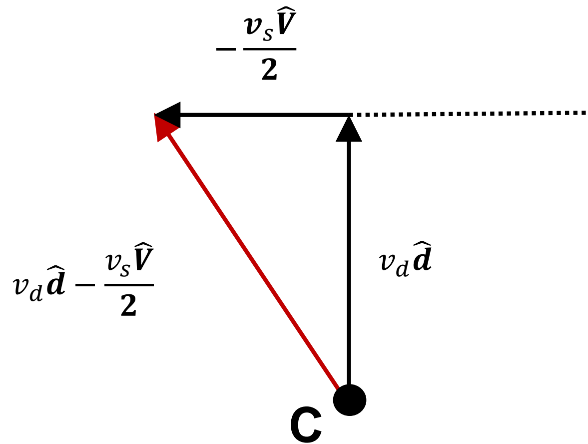

The initialized rays

The rays cast into the scene

The scene rendered to the viewplane

# Defining rays

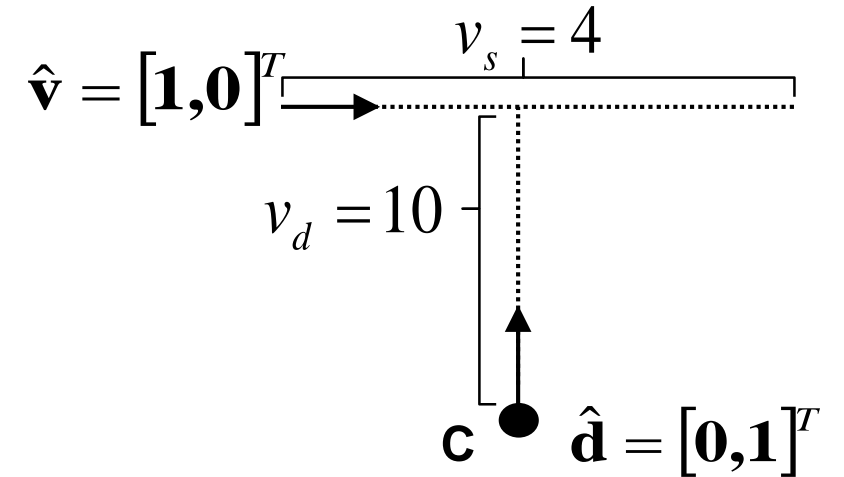
Recall from your lectures the definition of a ray:




$$\boldsymbol{R}_n = v_d\widehat{\boldsymbol{d}} - \widehat{\boldsymbol{V}}\left(\frac{v_s}{2} - \mathbf{n}\right)$$

# Lets try some exercises!

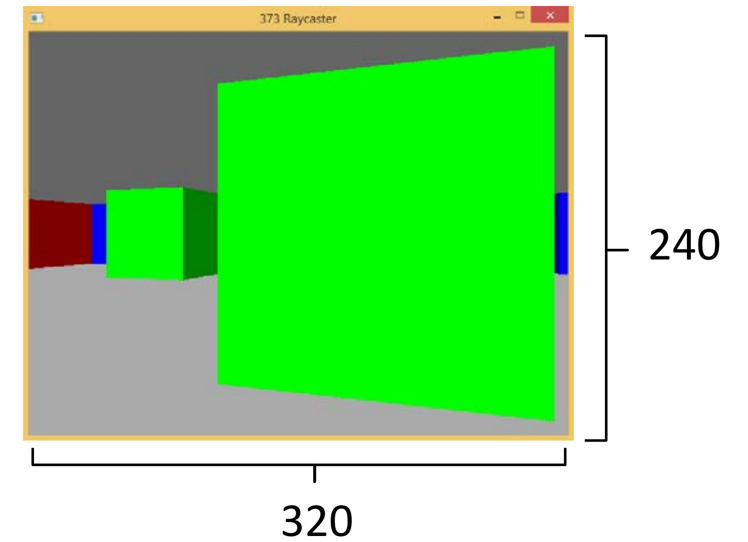
Assume that our projection environment looks like this:




1. What is the vector representing the first ray?
2. What is the vector representing the last ray?
3. What if $v_d$ was 6? What are these two vectors then?

# Lets try some exercises!

4. Considering the window to the right, what would $v_s$ be?

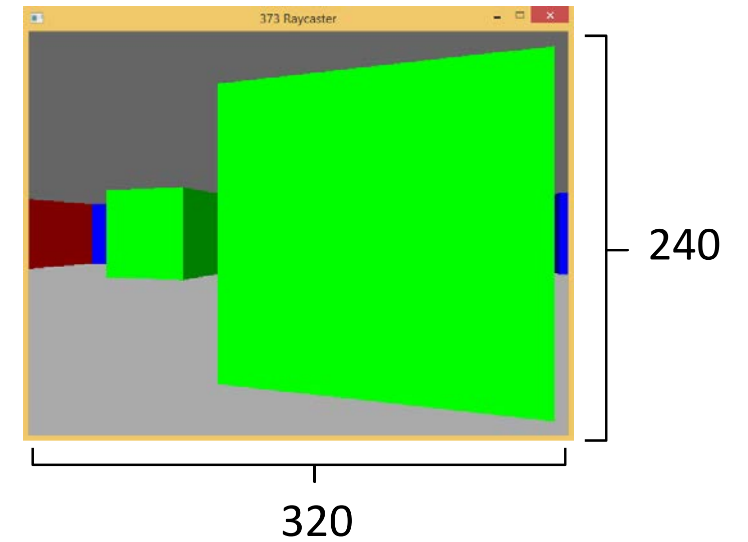



(Note: The window border is not included in the measurement)

# Lets try some exercises!

4. Considering the window to the right, what would $v_s$ be?

5. If we want a 60 degree field of view, how far away would the viewplane have to be from our camera?



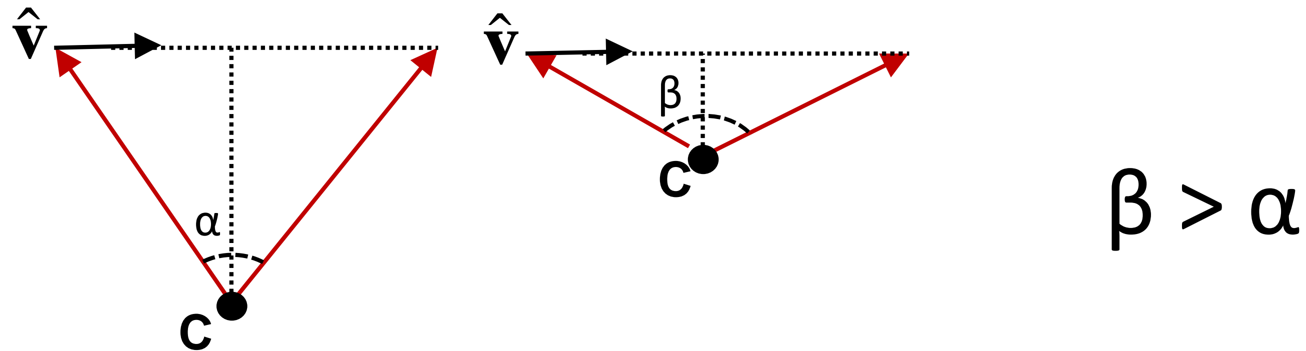

(Note: The window border is not included in the measurement)

# How do we calculate the FOV?

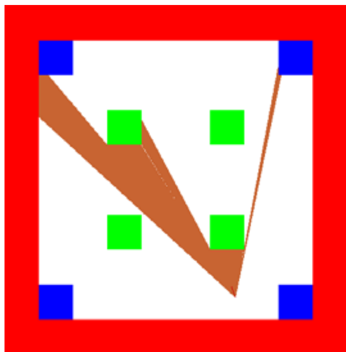
A camera's distance from a viewplane is directly related to the camera's Field Of View (FOV). The camera's FOV increases as the distance to the viewplane gets smaller, and decreases as the distance gets larger. This is analogous to looking out of the window in your house. The closer you are to the window, the more you can see!

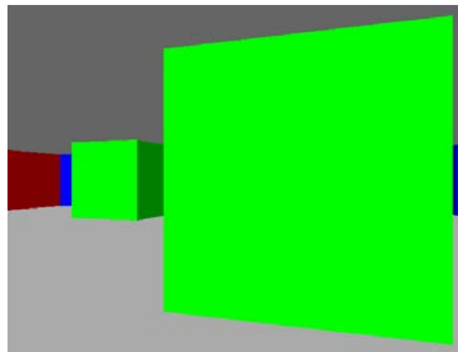
# Relationship to the view distance

Notice that that the FOV angle is much larger for a shorter view distance, than it is for a longer one.

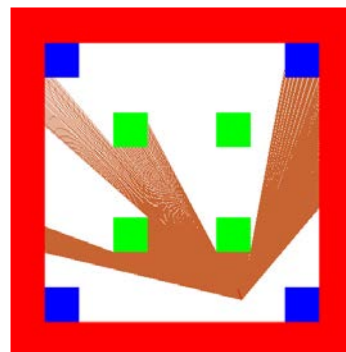


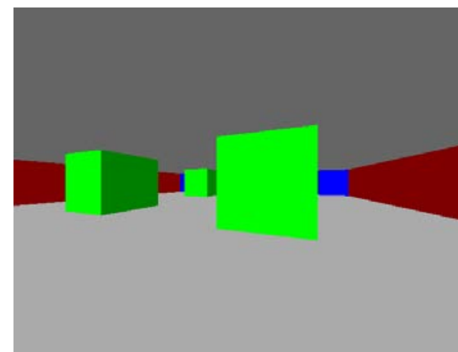

$\beta > \alpha$





A 'normal' FOV

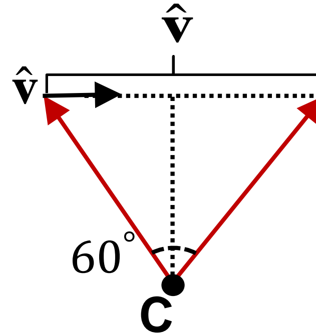




A 'wide' FOV

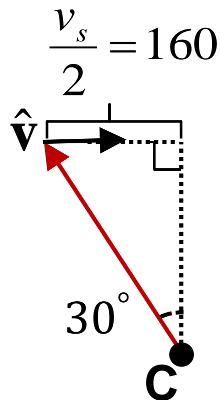
# Relationship to the view distance (cont.)

Lets take a look at what we have. Assuming out FOV is 60 degrees, and we have a viewplane length of 320, this would be our projection environment:

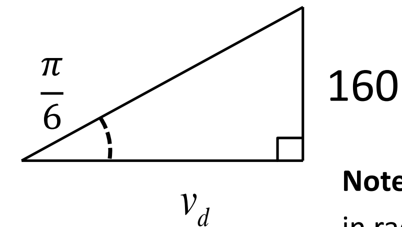



Because we need a right angle triangle to use our familiar trig identities, we split the environment in half:

$$\frac{v_s}{2} = 160$$



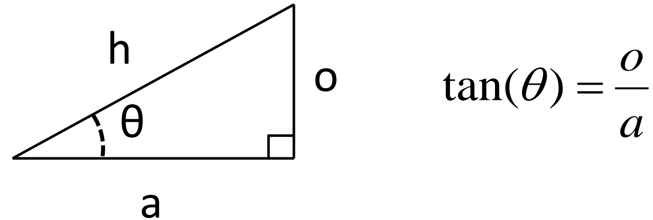

This gives the following triangle:



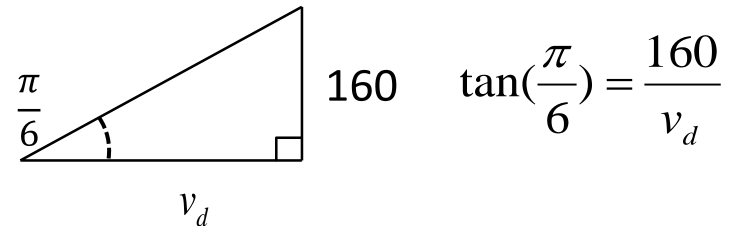

**Note:** $\frac{\pi}{6}$ is equivalent to 30° in radians

# Relationship to the view distance (cont.)

Recall the tangent trig identity:




In our case, this would mean that:




By rearranging this, we get:

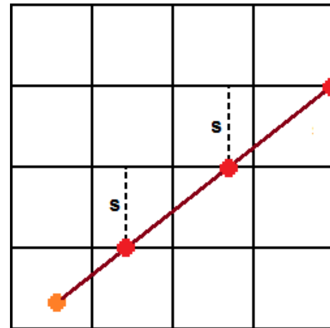
$$v_d = \frac{160}{\tan(\frac{\pi}{6})} \approx 277.128129$$

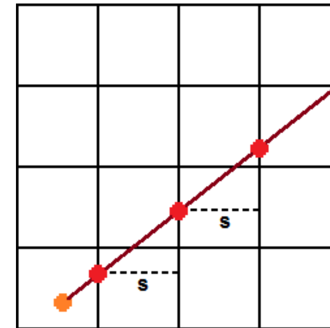
So, this means that our viewplane needs to be about 277.13 pixels from the camera for a 60 degree FOV.

# Casting rays into the world

Once we have our rays defined, we want to cast them into the world until they hit something.

As you've hopefully seen in lectures, we have quite a neat way of doing this. Instead of blindly jumping along our rays until we hit something, we use what we know about the world to make informed strides along the ray, only stopping at square intersection points.

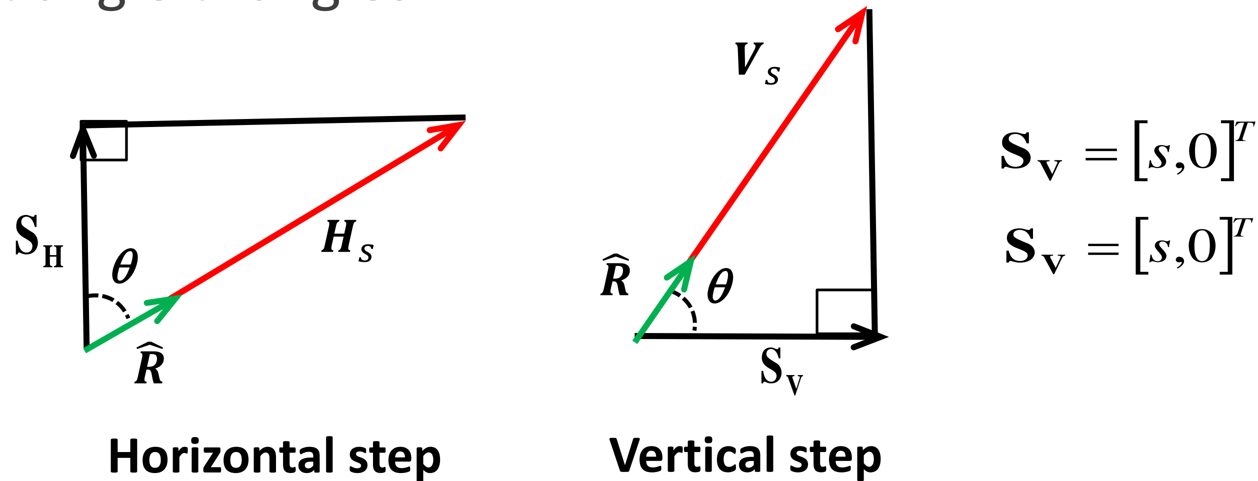



Notice that all the horizontal intersection points are equally distant from each other! Same for the vertical intersection points!

# Casting rays into the world (cont.)

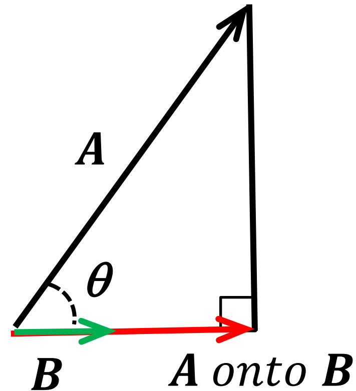
The steps between horizontal and vertical intersections can be represented by two similar right angle triangles:




This is great, because if we have any two components of a right angle triangle, then we can compute any other. In this case, we have theta and the adjacent side, and we want to calculate the hypotenuse (long side).

# Casting rays into the world (cont.)

Recall orthogonal vector projection:

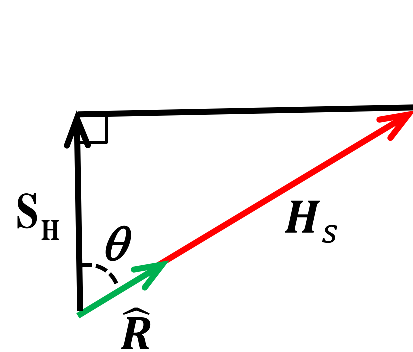



This is almost identical to what we want, except that it gives us the adjacent side of the triangle.

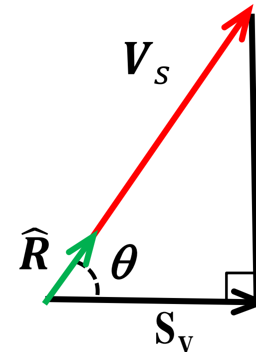
# Reciprocal vector projection

In order to get the side of the triangle that we want, we use the **Reciprocal Vector Projection** formula:

$$\mathbf{H_S} = \frac{\mathbf{S_H} \cdot \mathbf{S_H}}{\mathbf{S_H} \cdot \hat{\mathbf{R}}} \hat{\mathbf{R}} \quad \text{and} \quad \mathbf{V_S} = \frac{\mathbf{S_V} \cdot \mathbf{S_V}}{\mathbf{S_V} \cdot \hat{\mathbf{R}}} \hat{\mathbf{R}}$$




**Horizontal step**

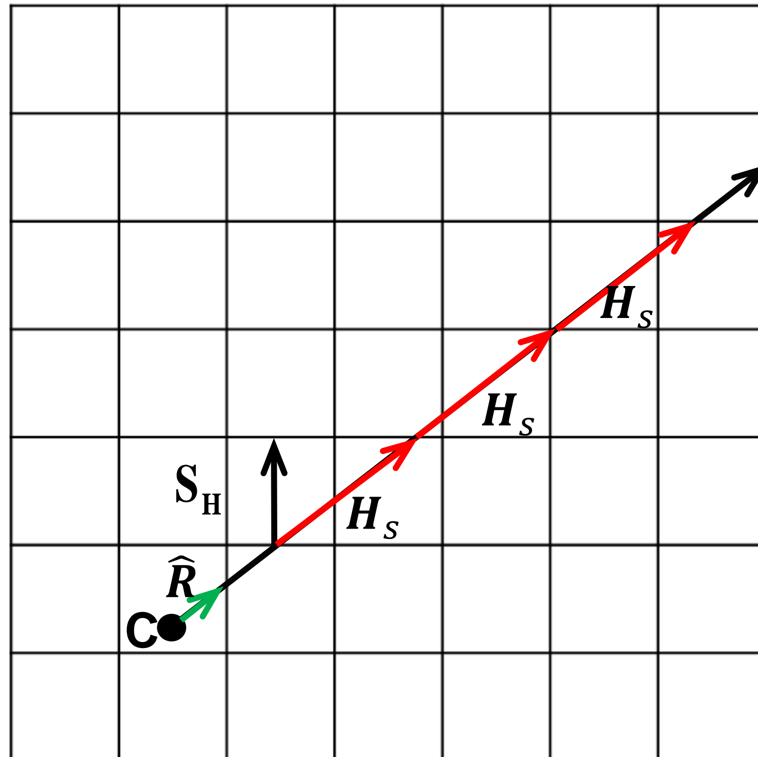


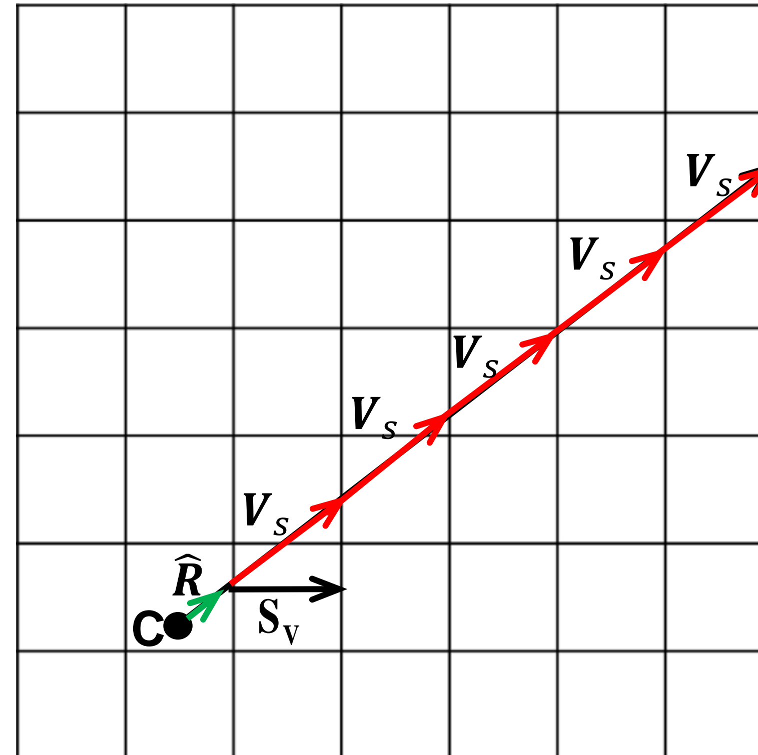

**Vertical step**

See me during office hours if you want to see exactly where this formula came from!

# Casting rays into the world (cont.)

Once we have $\boldsymbol{H}_S$ and $\boldsymbol{V}_S$, it's only a matter of stepping through our world with both of those vectors until we encounter a solid wall. The distance to the first hit still needs to be computed, but it's a similar process, and you can check your lecture slides for details.



That’s all for today!