Camera Obstacle Avoidance and Multiple Targeting

Alexandre Corcia Aguilera
Cs380

1 Introduction

One of the biggest problems when it comes to cameras in video games is that the players will only notice the camera if it doesn't work correctly meaning that the way the camera works needs to be perfect or almost perfect in order to have the user feel the best experience possible in the game.

So, how do we make the camera be as invisible as possible?

There are a lot of techniques in order to make the camera be more efficient or focus on different targets depending on what we want it to do but what happens when the camera suddenly can't see the player because there is an obstacle in the middle or when we need it to focus on multiple targets?

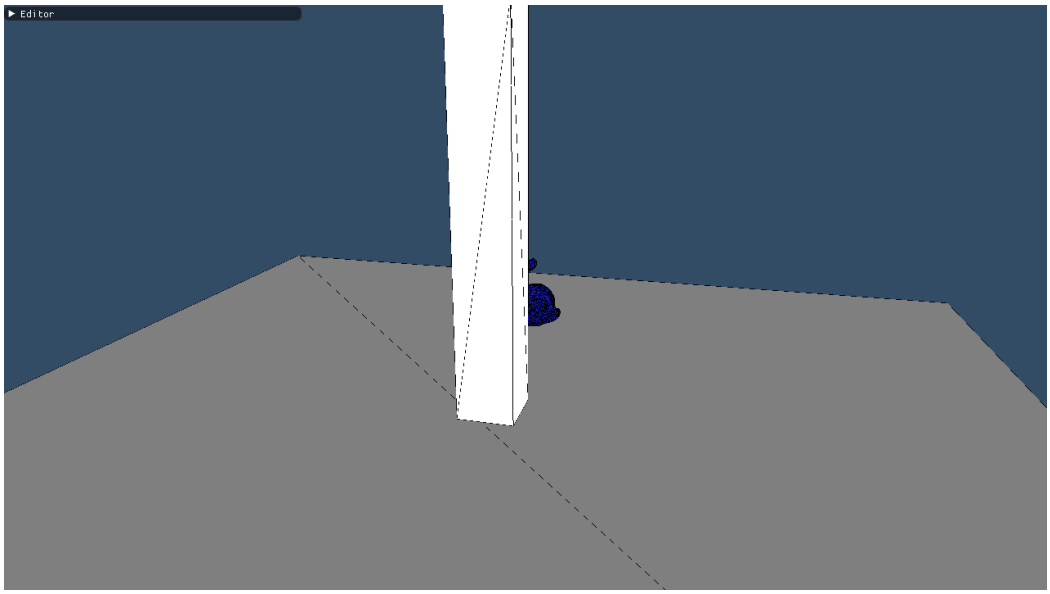
2 Problem Explanation in Depth

What problems are we trying to fix?

2.1: Obstacle Avoidance:

We can already see the problem in this picture. But there are more issues than the ones we can see in first hand.

Picture 1 Target is mostly hidden by an obstacle, this is what we wish to avoid



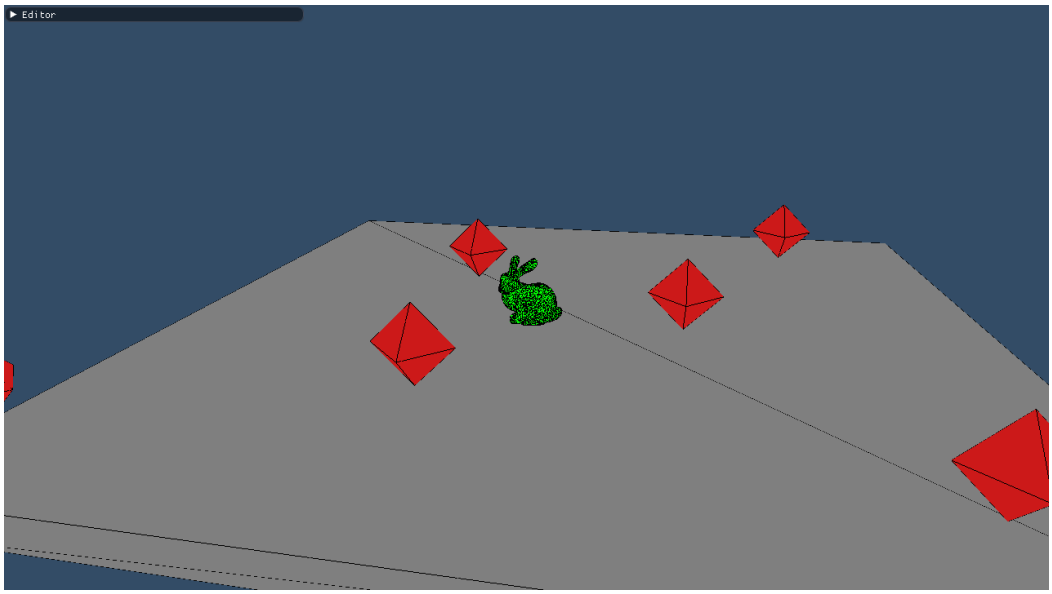
First of all we can't see the object that we are targeting meaning that the player doesn't know how what is going on.
Apart from that, how can we know if we are seeing the entire object or just part of it?
In the case where we are trying to solve this problem, what side should the camera go to?
How do we know what side is better left or right?

We will need to take all of this into account whenever we are trying to solve it.

2.2: Multiple Targeting

When it comes to having multiple targets, the main problem is not having any of the agents out of the camera or getting them out of the shot because we are trying to get other in.

Picture 2 Some targets are partially visible others are out of the camera's field of view



3 Detecting the Problem:

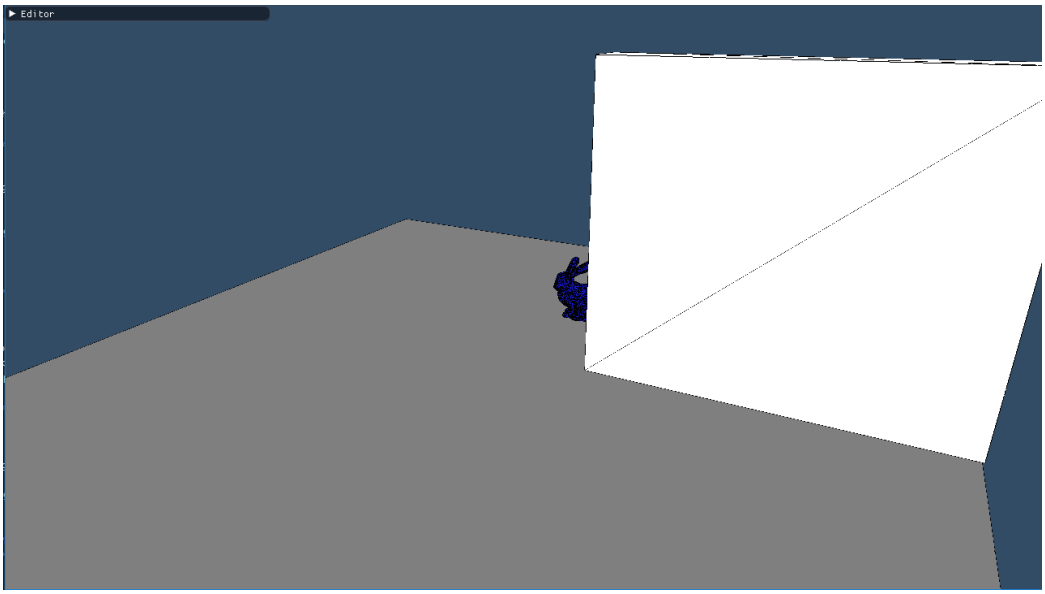
During this part of the document I will try to go step by step showing my walkthrough of the finding of a correct solution to the issues.

3.1: Obstacle Detection:

In order to check if the camera is able to see the object that we are targeting the first thing that we think about would be to use ray casting, throwing a ray from the camera to the position of the player. Normally if the camera is able to see the target the first entity the ray should encounter should be the target and if not there is an object between the camera and the target, so we should try to look for another angle right? Simple enough.

But let me present you a corner case:
What happens if the camera can only see part of the target but the center (the position of the target) can be seen by the camera meaning that the ray would indeed return that he is seeing the target but is only really seeing it partially.

Picture 3 The target is visible but right here the camera could take another angle so that the target is completely visible instead of only partially.



In order to avoid this issue we should send more than one ray:

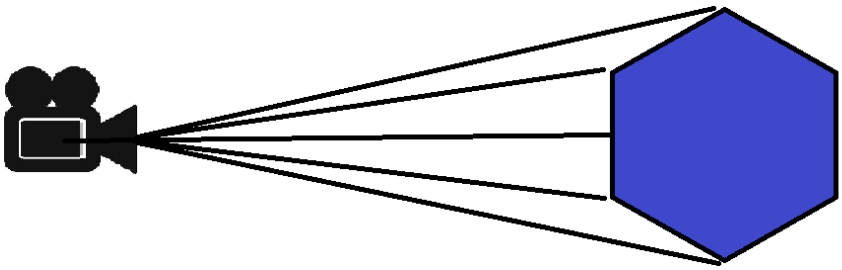
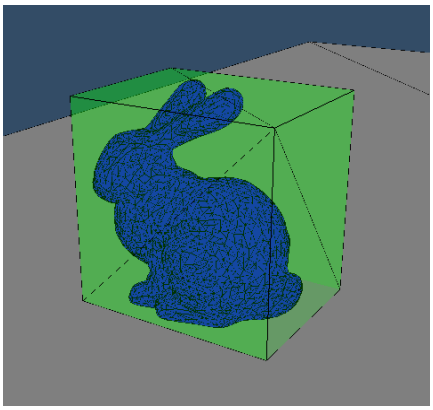


Figure 1 This is a representation of how we will be throwing the rays to the mesh that we are trying to target.

Doing this we are sure to check if the object is being partially hidden or not.
So we need to take into account all of the target's mesh but doing that with complicated meshes with a lot of different corners would be very expensive and not very profitable when it comes to computations.



Picture 4
The bounding box around our mesh
To avoid that, we will be using debug aabb (Axis Aligned Bounding Box).

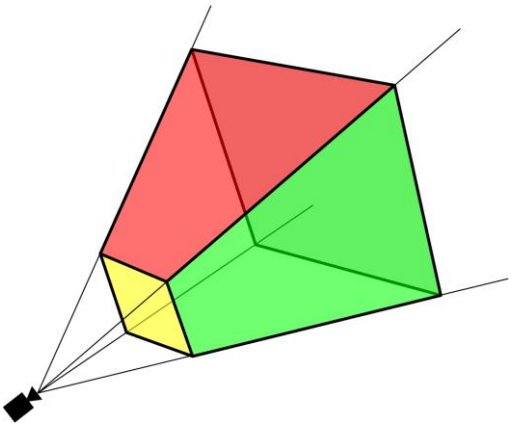
This will help us determining the boundaries of what we want the camera to see. And also lower the cost if we need to check the mesh every time we need to recompute a position because we know that the cost of checking the target will be limited to throwing 8 rays to each corner of the aabb and checking their results.

3.2: Multiple Targeting Detection:

When it comes to multiple targets, we need to take into account all of what the camera sees, meaning we can't assume that the camera will be looking at the direction of each entity so raycasting is not going to be a direct solution similar to obstacle avoidance.

We need to analyse what the camera is able to see, which means working with a frustum:

Figure 2
The frustum is geometrical figure that represents what the camera is able to see built with 6 planes. What the camera is able to see is represented by the colored parts.



We know that if an object is inside of the frustum it the camera is able to see it. So by using the same technique of box bounding that we used for the Obstacle Resolution Detection (see 3.1 in page 2), we will be able to check with the bounding boxes if the target that we are checking is inside of the camera's field of view (FoV).

4 Resolving The Problem

4.1: Obstacle Avoidance:

What do we do when we actually detect an obstacle ?

First we need to make the camera turn around the target, for that we will be using spherical coordinates in order to always have the same distance from the camera to the target and move around simply changing one of the angles:

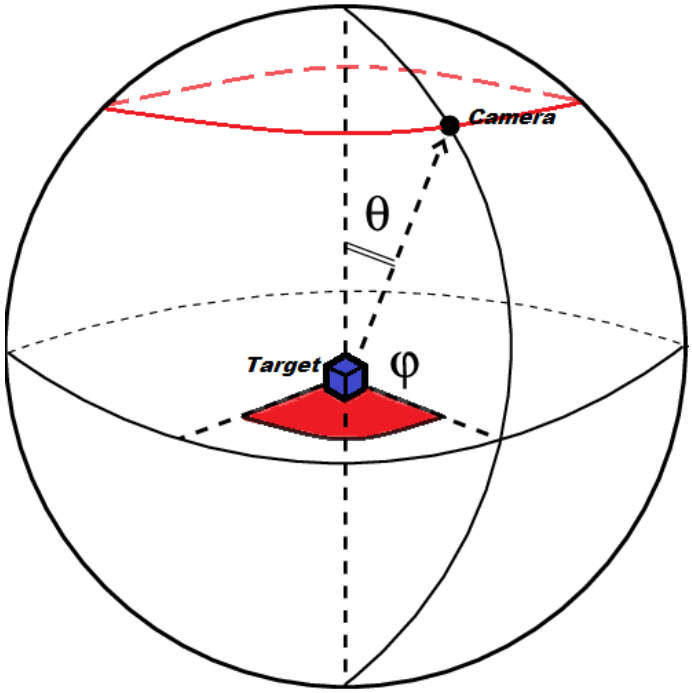


Figure 3

This are spherical coordinates, we will be using them because they are an easy way of keeping the distance (which will be the radius) from the camera to the target while at the same time moving the camera in a single spherical motion:

Spherical Coordinates:

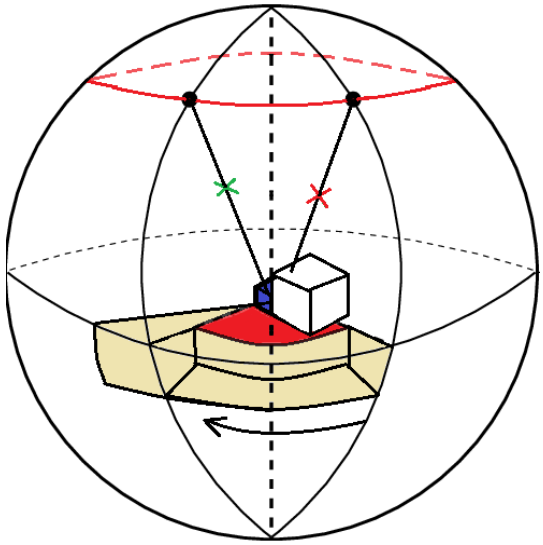
$$x = r * sin(\varphi) * cos(\theta)$$
$$y = r * sin(\varphi) * sin(\theta)$$
$$z = r * cos(\varphi)$$

$$r = \sqrt{x^2 + y^2 + z^2}$$
$$\varphi = arccos(\frac{z}{r})$$
$$\theta = arctan(\frac{y}{z})$$

To find a better angle we will be rotating around the vertical axis, meaning we will be modifying theφangle (or in the figure 3 above changing our position around the red line by modifying the angle in red).

Now that we know how to reposition the camera in order to find a better position, we will be checking in an angle variation the positions that we want the camera to look for a better angle, this also means that every time that we will be trying a new position we will have to redo the 8 rays to see if we are able to see the target this time.

Figure 4
This figure is to represent what angle variation we are talking about, the area in beige is the angle variation.
When the camera detects an obstacle it will look for a new position where the target is from that angle variation and if it finds one, the camera will have found a new position.



That is it, the camera found a new position where the target can be seen!
But could there have been a better one? Or maybe there is no position where the camera can fully see the target but there is one where it can see it more.

In order to take those conditions into account we will be quantifying the visibility of a new position in two ways:
Percentage of vision and distance from the original position.

Percentage of vision will be how many of the rays we are throwing are actually reaching the target. And the distance from the original position will be checking how much we moved from the first checking point.

The percentage of vision should be a value from [0,1] with 0 meaning that the camera is not seeing anything from the target and 1 being the complete vision of it

This will enable the camera to check both sides of the obstacle in order to see if it can find a new position.

Example of situation:
We are in the angle 0 (degrees) and we are not able to see the target, when we check the left side of the object we find a percentage of vision of 1.0 but then when checking the angle variation to the right we find a position which closer and also has a percentage of vision of 1.0 so we would choose right.

4.2: Multi-target Adaptation:

To see all of the targets we first need to find the best central point that will be the direction that the camera will be facing, this is done by computing the average position with all of the targets' positions.

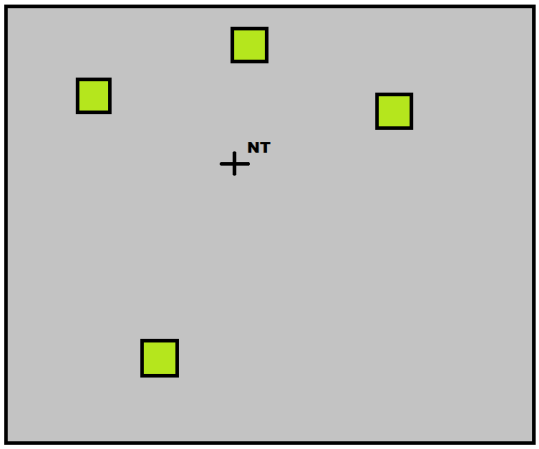


Figure 4 We can see here a bunch of targets and the computation of NT which would be the average of their positions.

New Target Position computation:

$$NT = (\sum_{i=0}^{n} T[i].Pos) / n$$

With n being equal to the number of targets
And T[] a container with all of the targets.

With the frustum and the aabbs we will go through all of the targets and checking if they are in the frustum.

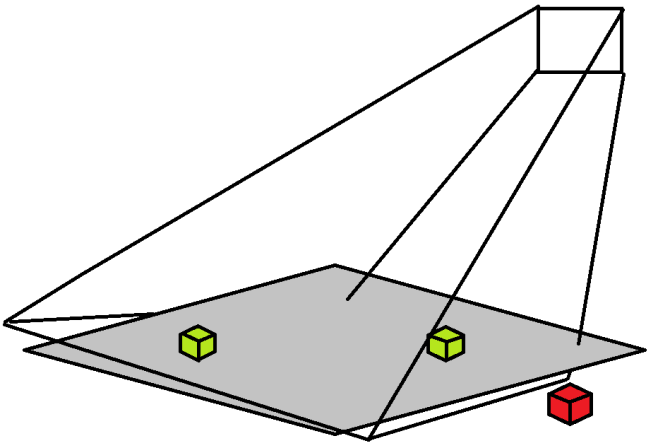


Figure 5 this represents the frustum checking all of the targets and finding which are in and which ones are out or colliding with it.

To adapt the FoV to see all of the targets there are a lot of variables that could be used, in this document we will be using the distance from the targets average point (DTA).

There are two situations where we would want the DTA to be modified, when there is a target that is outside of the frustum meaning that we would need to increase the distance until we would have all of the targets inside again, or when the distance is too big meaning that we need to lower that distance because we are wasting space.

The first case: where we want to increase the radius to have all of the targets in what we want to do is:

Pseudo Code:

```
*While all of the targets are not inside of the frustum
{
     *Compute a new position for the camera modifying the
     distance (adding distance).
     *Compute a new frustum with this new position
     *Check if all of the targets are inside
     {
          *If a target is out or overlapping
               *Start another loop from the while loop
     }
}

*If this point is reached it means that all the targets are
inside the frustum
*Change the frustum to the new computed frustum.
```

The Second case: where we want to decrease the radius to have a tighter Fov

Pseudo Code:

```
*Try a new distance that is closer
*Compute a new frustum with this new position

*Check all of the targets are inside
{
     *If a target is outside or overlaps.
          *Don't keep the new distance.
}
*If this point is reached it means that all the targets are
inside the frustum
*Change the frustum to the new computed frustum.
```

This will assure that the frustum that the camera is using is the best one for that FoV and that it has all of the targets.

4 **Improvements** (That could have been done)

There are many others thing that we can do when it comes to the camera in order to make it as invisible as possible to the player.

In project I felt like I could have done more, with some simple modifications if time wasn't so limited.

For example when it comes to the Obstacle Avoidance, a way of improving the algorithm could be to instead of checking only one angle in the spherical coordinates checking both of them so that we can also take into account that the camera can try to find the best position looking from a higher PoV.

Or another thing that in my opinion would have been a good addition for the Multi-Targeting would have been to give importance to the targets so that the camera would tend to look at a point closer to the most important targets.

There are other improvements that can be named, like Quadratic Interpolation to give the camera a smoother interpolation that is more appealing and that doesn't seem forced, that are not ai dependant but that make a camera shine in compare to other cameras.

5 Conclusion

In this document we showed how to do simple obstacle avoidance with spherical coordinates and a perception percentage method and also a multi-targeting technique that adapts to the positions and the separation of the targets.

These algorithms could be the basis of an intelligent camera but can be improved with other techniques where these fail like for example camera clipping into obstacles which could be done with collision resolution and would make a camera stand out

6 References:

6.1: Online Document:

Gamasutara : Real Time Cameras - Navigation and Occlusion
https://www.gamasutra.com/view/feature/132456/realtime_cameras__navigation_and_.php
by Mark Haigh-Hutchinson, July 1 2009

Notes from class
by Iker Silvano