Lode's Computer Graphics Tutorial

Raycasting IV: Directional Sprites, Doors, And More

Table of Contents

Back to index

Introduction

Raycasting IV is the latest addition the the Raycasting series, written in 2017, 12 years after the previous three parts.

For now, this part does not yet contain any example code, pictures or formulas, but high level descriptions of a few additions. This can allow to extend the code of the previous tutorials with these new principles. I hope this can be helpful in this form.

Vertical Camera Movement

In a raycasting engine, the camera typically cannot look up or down, only forward. We can only move horizontally, and rotate, which corresponds to changing camera yaw.

A few more things are possible with some extensions:

Changing pitch

Pitching the camera means looking up or down.

The effect described here is not perspective correct, since wall edges must remain vertical in a raycaster, but the effect looks convincing enough if the angle is not too large.

In the raycasting code, we calculate drawStart as "-lineHeight / 2 + h / 2" and drawEnd as "lineHeight / 2 + h / 2". This makes the vertical center of the wall exactly match the vertical center of the screen. To change the pitch, move everything vertically on the screen, that is, everything gets shifted up or down: add a value "pitch" to both drawStart and drawHeight, a positive value to look up, a negative value to look down.

Floors, ceilings, textures and sprites also need new handling, and are shown in the downloadable code example.

Changing the vertical position of the camera

Changing the vertical position of the camera is quite similar to changing pitch, but needs to take the distance of the wall into account.

Since jumping changes the vertical position, let's call the vertical position "jump". For the current vertical stripe, now we must add jump / perpWallDist to drawStart and drawEnd. To change both pitch and jump, add pitch + jump / perpWallDist.

Floors, ceilings, textures and sprites again also need new handling, and are shown in the downloadable code example.

Code for pitch and vertical position

The attached file raycaster_pitch.cpp contains everything from raycaster_floor.cpp and raycaster_sprites.cpp, but with both pitch and jumping patched into it. The variables pitch and posZ are new, you can search for all their usages in the code to find all the changes required for these features.

raycaster_pitch.cpp

A thank you goes to Michael Elliott for improving the formulas for changing vertical position.

Changing roll

In theory roll can also be changed. Then instead of having vertical stripes, the stripes are rotated with some angle and drawn rotated in the screen. The raycasting itself is identical, the only difference is how we draw the stripes on screen. For a 90 degree tilt this is easy, now we work with horizontal instead of vertical scanlines. For other angles, however, there will be an issue: when you draw each rotated stripe on the screen, you must make sure all screen pixels are filled, and it is likely they won't. Maybe making the lines thicker, e.g. repeating the same pixel on screen twice below each other, and then overlapping the duplicates with next lines will work, but I did not try it myself.

Directional Sprites

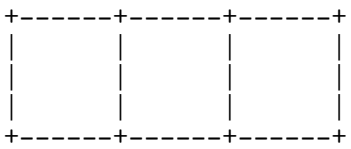
The previous tutorial introduced sprites. Those sprites had only 1 texture, and no matter from which angle the player looks at them, you always see the sprite from the same face, as if it is always looking towards you. This can be ok for enemies if you assume they always look at you, and for simple objects that look the same from all sides like round lamps, pillars and barrels or for objects put into an alcove so you can only see them from 1 side.

But if you want to be able to sneak onto enemies from behind, or have more complex objects that look different from different sides such as a chair or a tree that doesn't always look the same from each side, some better 3D illusion is needed. A raycaster cannot render such objects in true 3D, but we can step up things by rendering it differently when looked at from different angles. A typical amount of angles to support is 8 different angles. The object now needs 8 textures, 1 for 8 possible directions. If you design the object in a 3D program, you can save renderings from 8 angles with it. Or use good pixel art skills to draw an object seen from 8 angles.

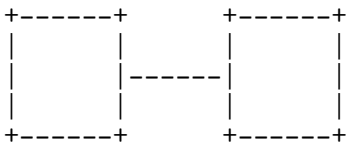
Then when rendering the sprite, you must choose which of the 8 angles to draw. This depends on the location of the player versus the location of the object in the 2D map. Calculate the differences dx and dy between player x,y coordinate and object x,y coordinate. Then take the atan2 of dx and dy to get the angle. Then round it to the nearest of the 8 supported angles, that is the index of the texture to choose.

Thin Walls

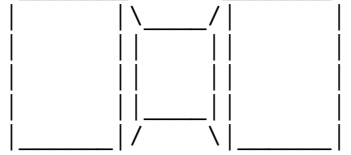
So far, the raycasting engine has focused on thick, square shaped walls filling up an entire cell of the 2D top down grid. For example here are three such walls as seen from above:



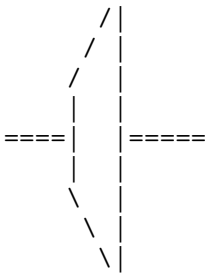
It is possible to make other shapes than a square, such as a thin wall halfway between the squares, such as the middle wall here:



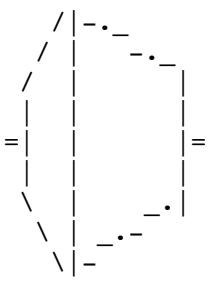
Again, that was rendered from above, from first person view of that thin wall would look something like this:



And if there were a free-standing thin wall, without the two thick walls next to it, it could look like this from an angle (with some distant walls === to complete the picture):



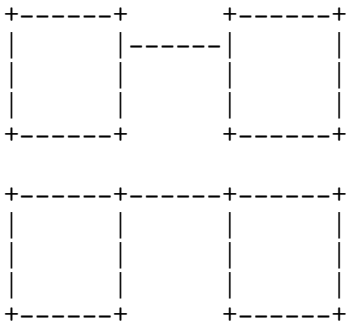
Which is quite different than how a free-standing thick square wall look from first person:



How to do it: Raycasting is 2D math, where a ray from the camera intersects a 2D shape, such as the square walls from the previous tutorials. While for the square walls we were doing such 2D intersection between the ray and the 4 sides of the square, now instead calculate only one intersection with a line segment halfway inside the square. In a sense it's simpler because it's only 1 instead of 4 line segments. There are two types to take into account: those in east-west direction, and those in north-south direction.

With tiles like this, either the ray will hit the thin wall inside, or it will not and go past it. If the ray does not hit the thin wall, then we don't do anything with this tile and continue the ray using DDA steps to the next tile.

The thin wall doesn't have to be in the center, you can make other variants, such as shifted at other positions than the center:



Doors

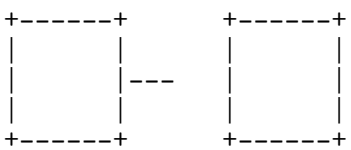
Doors can be done similarly as the thin wall. However, this time we also want to render the sides of the door (optionally), and have the ability for open/closing animation. Rather than a single line segment - or | like the thin walls, now we have a H shape, where the two vertical bars of the H represent the two sides of the door.

For the opening/closing animation, a timer is needed. A simple way, is to have an extra 2D map, containing a floating point timer for every single cell, and yet another map containing an integer with possible codes 0 (closed), 1 (opening), 2 (open), 3 (closing). Only those for doors will actually be used of course, this just makes it easy to address. A much more optimized game like Wolfenstein 3D did not work like that but had, as far as I know, 64 timers with individual doors pointing to them, and so didn't support more than 64 doors in the whole map.

Then read a key such as space to detect player opening a door. Calculate if the square in front of the player is a door and its coordinates. Then activate its timer: update the special integer code to become 1 (opening) if it was closed, or 3 (closing) if it was open. The floating point timer value can be 0 (closed), 1 (fully open), or anything in between (e.g. 0.5 means it's halfway open). But don't update this one yet when the player presses space.

Then every frame, go through all the tiles, and for every tile that is a door and has special integer code 1 (opening) or 3 (closing), update the floating point value: calculate the time in seconds between this and last frame. Based on the time difference, add (if opening) or subtract (if closing) a small value from the floating point value. If smaller than 0, cap at 0 and set special code to 0 (closed). If larger than 1, cap at 1 and set special code to 2 (open).

Finally, for drawing the door, use the timer value to determine line segment length, which is 1.0 - timer. For example if the timer is 0.5, the situation looks like this:



Intersect with this dynamically smaller line segment (using the timer for its length), and also subtract the timer value from the texture X- coordinate so that the texture itself slides too.

Secret Push Walls

Secret push walls, as seen in Wolfenstein 3D, also move, similarly to doors. Except this time we are not moving a thin wall and its texture left or right, instead we are moving an entire block. We can reuse the same timer here. And, just like in Wolfenstein 3D, we are constrained to rendering something inside of the current grid square, so the secret cannot truly move out of the box, we have to fake it by moving only the visible sides of the wall.

To understand how the 2D square walls from the previous tuturials are truly drawn, this here shows the reality, in top down view (not first person):

```
Intended shape:
+------+
|      |
|      |
|      |
+------+

Actual shape:
       +
       |
       |
       |
+------+

         * we stand here
```

As you can see, we only see the two sides of the square that are on our side, the back ones don't exist. The reason I show this because it helps understand how Wolfenstein 3D "cheats" for drawing the secrets. It goes something like this if you have a free-standing secret rather than one properly put between other walls in Wolfenstein 3D (except it's possibly even glitchier IIRC):

```
t=0s
       +
       |
       |
       |
+------+

         * we stand here

t=0.5s

     +
     |
     |
+-------
     |

         * we stand here
t=1s

   +
   |
+-------
   |
   |

         * we stand here
t=1.5s

  +
+-------
  |
  |
  |

         * we stand here
t=2s

+-------
|
|
|
|
```

```
         * we stand here
```

And that is why a secret should be put between two other walls. The drawing below shows what the illusion above then looks like for us, with the back walls removed again, as only the front is what we see (shown in top down view, not first person):

```
t=0s

       |      |
       |      |
       |      |
+------+------+------+

          * we stand here

t=0.5s

       |      |
       |      |
       |------|
+------+      +------+

          * we stand here

t=1s

       |      |
       |------|
       |      |
+------+      +------+

          * we stand here

t=1.5s

       |------|
       |      |
       |      |
+------+      +------+

          * we stand here

t=2s

       +------+
       |      |
       |      |
       |      |
+------+      +------+

          * we stand here
```

Now it looks good. We will do something similar to Wolfenstein 3D, except we will draw only 1 thin wall instead of 2 to avoid seeing a wrong one. So we still need to ensure to put the secret between other walls, if we don't, then when the player activates the secret it will look as if our solid square suddenly turned into a moving thin wall (you can fix that too if you want, by rendering a solid block that becomes thinner on this tile and in addition rendering a matching solid block becoming thicker on the next tile, this will be a bit more work and some more messing with timers and special codes).

To implement it: you can reuse the same timer buffer and special code buffer as from the doors. Put some special value like 4 (0-3 were already for doors) in this buffer to indicate a secret wall. When the player presses space in front of the secret wall, it becomes a moving secret wall, in 4 possible directions depending on from which side the player pushed. So give e.g. special codes 5-8 for that. When it is moving, similar as

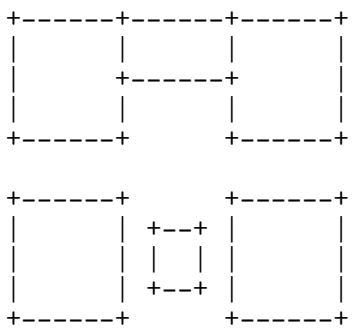
for the door, update the timer every frame, and use that timer to give the correct 2D intersection coordinates to a thin line segment. Now it does't move sideways but backwards. Depending on if the special code is 5, 6, 7 or 8, draw one of 4 possible thin walls, from North, East, West or South side.

A secret has to move across multiple walls, e.g. 2, otherwise it will not allow the player to pass. We could make the secret keep moving forever until it hits a fixed wall. How to do this: Once the timer of the secret on this tile reached the end, clear this tile, remove its wall, it becomes empty space. If the next tile (using the correct one given our distance) is free, turn that tile into a secret instead. Set its special code to the same one we had here, 5-8, and begin its timer. And so on...

Other Wall Shapes

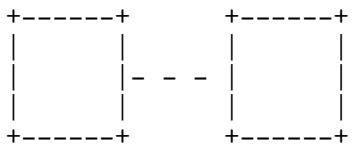
If you really dig the 2D intersection math, you can also make other wall shapes. All these shapes are shown top-down here, while in the game you see them from first person perspective from the side. All drawings show two regular square grid walls to show the grid, with the special shape in the middle.

Idea 1: half-sized or smaller walls.

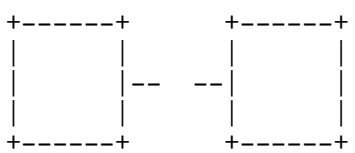


These can be implemented in a way similar as the big ones, except the intersection happens with smaller sides. If the ray does not hit the small part, it continues to the next tile in the DDA.

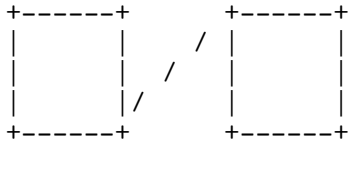
Idea 2: fence from vertical bars.

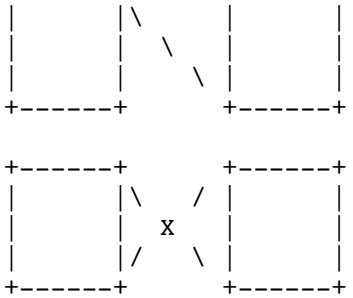


This is similar to the thin wall, except we have not 1 big line segment, but multiple small ones. Do the intersection with each of the small ones individually. If none is hit, then the ray continues throug the fence to a further wall in the distance. Make a matching texture for this, and you have a neat type of see-through wall that doesn't need a Z-buffer. A somewhat similar idea is a slit:



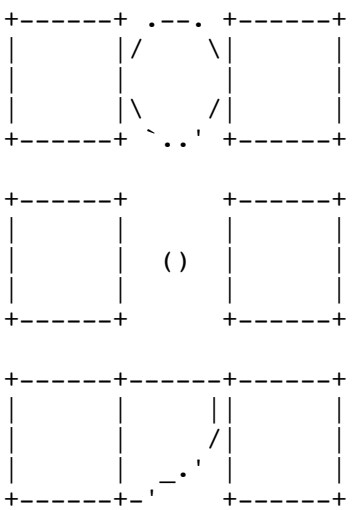
Idea 3: 45-degree angled walls.





Such wall allows for non-orthogonal walls. There is 1 extra complication here: so far we always had only north-south and east-west walls, and we gave two different brightnesses to distinguish the two. We now need to choose two more brightness variants for those two possible directions of 45-degree walls. When rendering this texture, apply some in-between shading.

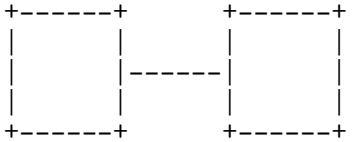
Idea 4: round pillars



For this, a 2D intersection between line and circle (or circle segment for the 1 quarter example above) is done, the formula can be found e.g. on Wikipedia. Then, to choose the texture coordinate, calculate the angle between the intersection point and the center of the circle using atan2. For the shading, even more shades than for the 45-degree slanted walls are needed now, the more the smoother the circle effect looks. Remember, to make a color darker, multiple its R, G and B coefficients with some floating point number in range 0.0-1.0, the closer to 0 the darker.

Transparent Walls, Z buffering

Similar to sprite textures from Raycasting III, transparent walls can have such textures too, best done for thin walls, as with thick walls it would look weird because we don't render their insides, which a transparent texture would reveal.



For this, while casting the ray, we will hit this transparent wall. Rememember its distance from the player. Then continue casting the ray to next tiles as usual. If we encounter more thin walls, remember those as well. Finally stop when hitting a regular non-transparent wall, as usual. Then draw the stripes we remembered in reverse order: first the regular wall, then the earlier ones in reverse order. This is very similar to the painter's technique used for the sprites in Raycasting III, where they were sorted them from farthest to closest.

To take into account sprites that can be in-between transparent walls, we need to use the ZBuffer from raycasting III as well. For the sorting, though, a merge is needed: sprites were sorted beforehand, but the

transparent walls are only known for this particular stripe we are raycasting. We already have them in the correct order though (in reverse order as mentioned),. So in O(N) time, you can merge the transparent walls in between the sprites. Search the internet for a merge algorithm if desired.

To circumvent the sorting altogether, both the sprites case and the thin wall case can also use a 2D ZBuffer instead, that is simpler, just uses a bit more memory.

So instead of having storage for each vertical sprite in a 1D buffer, store a distance for every pixel of the screen in a 2D buffer. Whenever drawing a non-transparent pixel of a sprite, transparent thin wall, as well as any regular wall, store the draw distance in the Z buffer. Only draw if the distance is not bigger than what is already stored in the Z buffer, that means a closer object was already drawn at this pixel. Then walls, transparent walls and sprites can be drawn in any order you want, the closest ones will always be the final ones written to the pixel.

Don't forget at the begnning of every frame to clear the Z buffer again (set all values to a very high or infinite value).

The 2D Z buffer is not easily compatible with the semi-translucency however, so if you'd like that, go for the sorting scheme mentioned above. You can still do something like it by remembering multiple distances and the sprite or wall it belonged too and then rendering in correct order per pixel or stripe at the end, it's a bit more complex and slower, you could remember some limited amount (e.g. 4 to see up to through 4 translucent things), or make it unbounded (probably slower).

If there are no sprites, we could also support transparent walls without any Z buffer, since while ray casting we go through them in inverse order, then just invert the order at the end and draw them one by one. However, that is not compatible with having sprites, as there may still be sprites in between there. So the most universal way is to use a ZBuffer for everything.

Animated Textures

Animated textures are time-dependent. If you have multiple frames for 1 texture, then determine which frame to choose based on time. Use the time in (milli)seconds, not the amount of frames rendered, to ensure it's consistent no matter what frame rate.

A different way to animate is to have only 1 texture, but move it around. It could for example be moving in a constant direction, or wave around in X and/or Y direction using sine functions. This is nice for a water or lava texture on the floor. Here, then, we use time to choose the relative coordinates of the texture, rather than the texture itself. That is, when hitting the wall/floor/ceiling and calculating the texture coordinate, after that add the dynamic value to it, and use modulo to make the texture tile (for the modulo, you cannot easily use the % operator or fmod function of C/C++ as they behave wrongly for our purpose on negative numbers, so when using integers, mask with texture size - 1, assuming it's a power of two, or better looking, using floating point numbers to allow pixels themselves to move smoothly, add texture size if smaller than 0, or subtract texture size if higher than texture size), then render.

Fog

Fog is relatively easy to add and looks nice: mix the pixel colors being drawn with a fog color. The bigger the distance of the stripe, the more the fog color dominates. Use a weighed average to mix the colors. E.g. to have 75% fog, then there is 25% color, so use (1 - 0.75) * color_r + 0.75 * fog_r for red, and idem for green and blue. The further away the stripe, the bigger the fog percentage and smaller the color percentage.

Black as fog color works for emulating night with limited viewing distance, white or grey looks like daylight fog, green can give a poison cloud look, red a firey look, etc... You can also make the fog local by storing amounts of fog in a 2D map and only rendering it when the player is on such fog tiles. This is of course not true volumetric fog, that is, the player can look right through a fog location when standing on a non-fog tile, but it already gives nice local effect and the issue can be hidden by having the fog in local rooms obscured by hallways in between.

Enemies

To make a true raycasting game, enemies or NPCs are needed. These are rendered very similarly to sprites, but they can move around freely. So unlike sprites, they are best not stored as a 2D grid map, but as their own individual entities. They must also be able to move smoothly, so are able to stand on any location, not just in the exact centers of grid tiles, so they need floating point coordinates. So other than being at 0.5 positions, rendering them is otherwise exactly the same as for sprites, with the same Z buffer and sorting principles (or no sorting if using the simpler 2D zbuffer), they just are gotten from a different list than the regular sprites, the list of enemies.

If you would like to use directional sprites for enemies, then two angles must be taken into account to determine which texture to choose to render: the angle between player and enemy on the 2D map, and, the looking direction of the enemy (note that looking direction of player does not matter).

Enemies also need AI that makes them move around, several states (such as waiting, alert, dead, ...), mechanics such as hit points, and so on. This tutorial is about graphics so will not delve into this. It is where the engine starts to be a game rather than a graphics engine though, so it is a very worthy subject to look into. We will still delve a little bit more in gameplay aspects with the next section about weapons.

Enemies likely need animated textures, such as when walking around, shooting, etc... See the animated texture section for this. Depending on the state of the enemy, different animations and texture sets are chosen.

Weapons

To further make it like a real game weapons may be desired. There are a few types of weapons possible in a raycasting engine:

- Direct-hit weapons
- Projectile weapons
- Melee weapons

For direct hit weapons, you can reuse the raycasting code to detect if your weapon hits an enemy. To do this, some refactoring of the code from the earlier tutorials is needed: extract out the part of the code that calculates the intersections of a ray from a starting point until it hits a wall (or multiple walls if you support transparent walls). Also extract out the part that calculates if a sprite is in view. When shooting a direct hit weapon, do this raycasting calculation for the center stripe of the screen (as you point the gun at the center), and get the closest thing, wall or sprite, that was hit. If it is a sprite, and that sprite is an enemy, then it gets hit and receives the hit event (so it can lose hit points, have its AI respond to it if needed, ...).

For projectile weapons, shooting the weapon causes a projectile to spawn. Similar to an enemy, this projectile is a sprite, drawn like a sprite, that moves around according to some rules, this time not according to an AI, but a simple movement in a constant direction with some constant speed. At some point it hits something or disappears, so the object list (and if used, sorting) must be updated. If it hits an enemy (which can be detected by having its distance from the enemy smaller than some floating point value), the enemy loses receives the hit event. You can make projectiles that bounce on walls, change direction, guided missiles that track enemies, and so on, too.

For melee weapons, this can be as simple as calculating how close you are to an object in front of you, that is, for every object, calculates its distance and angle from you, if correct angle (in front of you) and close enough, it's hit. For a longer melee weapon, like a polearm, you may need to use raycasting like for a direct hit weapon instead, or else it might work through walls. This is very much the same, except if the closest distance is too far, it has no effect.

Portals and Mirrors

It is possible to create portals that you can see through, by having the ray, when it hits the portal, continue tracing at a different location, the destination of the protal, and then the raytracing continues just as usual from there. To make it functional, when the player walks up to it, the player too can teleport to the destination. The sorting of sprites by distance is no longer relevant here, so make sure to use a 2D Z-buffer now, or store sprite sort orders per portal (probably more complicated than it's worth).

Mirrors can be done in a very similar way, here the ray changes X or Y direction depending on which direction the mirror is in.

In both cases, putting a transparent texture in front of it can improve the effect or reduce some visual confusion, such as some borders around the portal or mirror or add some imperfections.

Collision Detection

The raycasting programs in this tutorial have very primitive collision detection when moving: all it does is check if the spot you move to has a wall or not. If there would be two diagonally adjecent walls touching only by the corner, if you aim your movement just right this could allow moving through the wall at that spot.

A possible solution for this is to use the raycaster itself for collision detection: rather than only check if the destination point of the movement is wall, raycast a line from the player position to the next position. If it reaches the point, the player can move there. If the ray hits a wall before that point, that's the final point where the player movement can end up.

To avoid awkwardly close to the wall camera views, the player movement is best stopped a bit before the collision point, rather than at the wall itself, so the ray can be extended a bit further to test for distance to wall.

Raytracing

Raytracing is an extension to 2D and 3D of what Raycasting is in 1D and 2D. That is, for raycasting, we cast 1 ray per 1D vertical stripe and do 2D intersection math with objects. In raytracing, instead you cast 1 ray per pixel of the entire 2D screen, and do 3D intersection math with objects.

To have a form of the raytracing very similar to the raycasting engine here, you could have a 3D map existing out of voxels, where every voxel is a cube with a texture on the 6 sides. This is very similar to the raycasting square walls with textures on 4 sides. The ray goes through the 3D voxels very similarly to how we go through 2D squares. When a voxel is hit, calculate the 2D coordinate on the side of the cube to get the texture coordinate, then give this pixel the color matching that texture pixel.

However, raytracing usually does not use such voxel map at all, but instead there is a list of objects, such as spheres, triangles, etc... Then we have to calculate for each of those objects, whether our ray intersects them. Then for the closest object, calculate the texture coordinate and draw that as our pixel.

There can be many triangles, for complex polygonal models, and in theory we have to intersect with every single one. To speed things up, we can avoid having to calculate all, by such techniques as dividing the space in big voxels, remember which objects are in which voxel, and only intersect with those, or binary space partitioning, and other techniques. Also, for objects existing out of polygons, have a bounding box, and only calculate individual polygons if the ray goes through the bounding box.

For translucency in raytracing, keep the ray going after hitting translucent objects and remember each you encounter. Then render them in reverse order, similar to the translucent sprites from Raycasting III.

For mirrors and refraction, change the ray direction according to optical formulas as it hits such surfaces.

For light sources, after our ray hit an object, then draw a ray from that point to all lights. If the ray is not blocked by something between the starting point and the light, add the color of that light to this point, with

brightness depending on distance, and e.g. using shading such as Gouraud shading or Phong shading instead of simple flat shading on polygons for good looking effect.

There are much more complex raytracing techniques to support, for example, indirect lighting. We can also raytrace the other way around, starting from light sources rather than from the camera, or start from both and meet in the middle. Look for global illumination and The Rendering Equation.

Last edited: 2020