Rasterizer Project

Adrien GIVRY & Hanseul SHIN

1. What is the rasterization?

Before jumping into this broad idea, let’s go back in the time.

Think about the time where people had no access to the internet, the computer and

the television, where the night was dark, and the morning was bright. Working in a

farm in the morning, pray and sleep with your family at night is all you can do.

One day, a man decided to open his Bible and saw

*“And God said, "Let there be light," and there was light.” Genesis 1:3*

The light, this is where all began.

1. The light

“**Light** is electromagnetic radiation within a certain portion of the electromagnetic

spectrum. The word usually refers to **visible light**, which is visible to the human eye and

is responsible for the sense of sight.” [1]

René Descartes is the one who started the modern concept of the light.

He said that the light is a mechanical property of the luminous body. Later, this idea

became the principle concept of the Particle theory. Isaac Newton believed that the

light was emitted in all directions from a source and the light was moving in a line

form, not in curve form.

This theory led the humanity to the concept of the camera.

1. Camera Obscura

“Camera obscura (pinhole image) is the natural optical phenomenon that occurs

when an image of a scene at the other side of a screen (or for instance a wall) is

projected through a small hole in that screen as a reversed and inverted image (left

to right and upside down) on a surface opposite to the opening.” [2]

This is the first camera which projected the light through a pinhole in the center and

make the light draw the inverted image on a surface.

Ex.

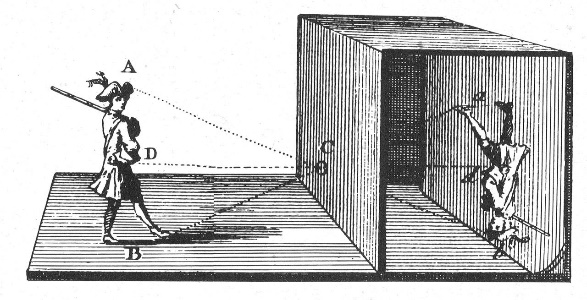
If we took this image of zebra with the pinhole image



The result is the image below.



For further explanation[2]



1. The screen

The concept of camera obscura is also used in the modern computer screen.

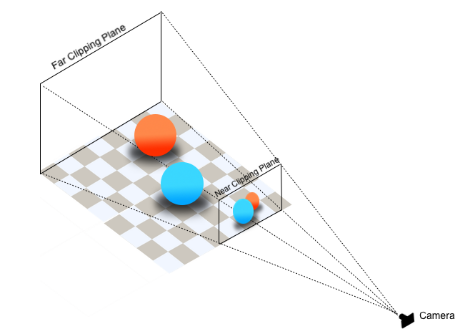
The side, the position and the angle of the object will depend on the position of the

Camera, the object and the screen.

If the image is closer to the screen, the image representation will also be bigger.

If the distance between the image and the camera is smaller than the distance

between the camera and the screen, the image won’t be projected to the screen.



Light traveling from the object will go through the screen and will reach the camera.

When the light reaches the screen, the position of the light will be the

The coordinate of the object in the screen.

1. Graphics pipeline

Until now, we saw the basic concept of the light and the camera.

Now, I must explain what the graphic pipeline is.

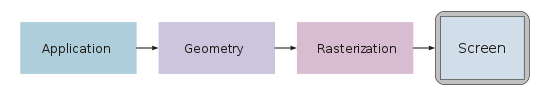
The graphics pipeline is the descriptions of what steps a graphics system needs

to perform to render a 3D scene to a 2D screen. We can easily calculate the

coordinates of a 3D object but showing it in a 2D screen is a problem.

Multiple steps are required to perform this, and it must be processed in the right

order.



1. Application stage

The application process is where everything is prepared by CPU.

Let’s say you want to draw a cube.

The cube is placed at the origin (0,0,0) and the length of a side is 1.

So logically, the coordinates should be

Point A at x = 0.5, y = 0.5, z = 0.5.

Point B at x = 0.5, y = -0.5, z = 0.5.

Point C at x = -0.5, y = -0.5, z = 0.5.

Point D at x = -0.5, y= -0.5, z = 0.5.

Which will make a square.

And rotate this square on the axe of y or x by 90 degrees to make 6 faces.

These point coordinates will be called the “Vertex” / “Vertices”.

The problem rises at this point: The CPU have no idea what to do with these

vertices.

So, you must specify how you want to draw those points.

In other words, you must sort in which order you want to draw the coordinates.

This order is called the “Primitives”.

On each vertex, we can set the color data, which will be later used in the

rasterization stage.

With Vertices and Primitives, the CPU can determine how and what to draw.

To draw a cube, CPU will draw lines to make the triangles on each face.

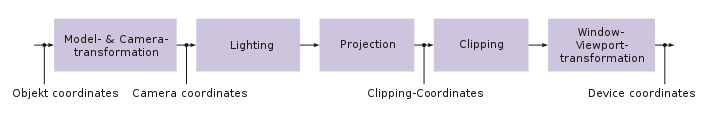
Unlike other forms of polygons,

with given angle, translation, scaling or any means of transformation,

the triangle stays the same. The triangle will continuously be the triangle.

This is the reason why the 3D graphics developers chose to use the triangle.

1. Geometry stage



Now all the polygon lines are ready to be drawn.

But the problem is, the coordinates of all the objects are not in 2D screen space.

Since a screen can only use X and Y coordinates, the cube at this stage

(with z coordinate) will be drawn as a pixel sized square.

So, we must “transform” these coordinates to the screen space coordinates.

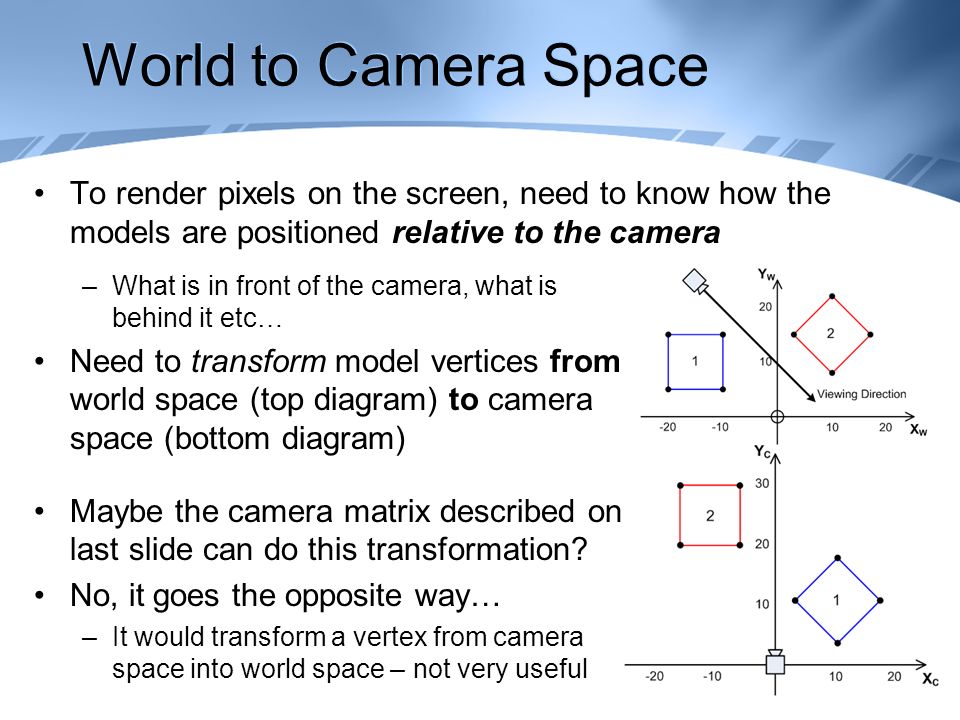
b.1 Object space to Camera space

This is where the fun begins.

The current coordinates of the cube might not be seen by the camera.

So first, we need to transform its coordinates to the camera space

coordinates.



Unlike some complicated cases, our camera is fixed at the origin (0,0,0) so, all we

need to do is applying translation, rotation and the scaling using the matrices.

b.2 Model-View matrices:

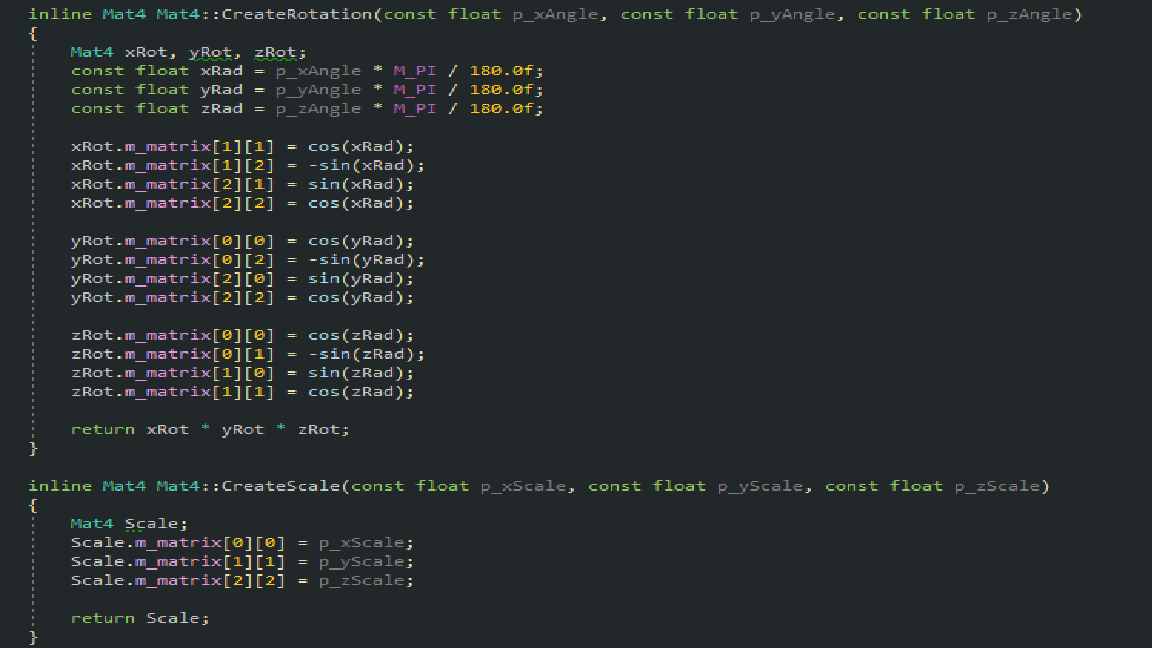
To transform the object from the “world” space to the camera space, we need to

apply the model-view matrix.

We Scale the object, we rotate the object and then we translate the object.

The order is very important, since rotate and translate will not give the same

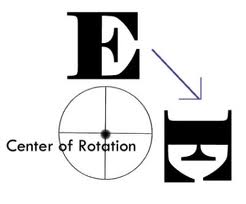
result as translate and rotate.



This is our rotation and scale matrices.

We calculate rotation per axe and multiply them together.

Ex. Rotation of letter E.



b.3 In camera space

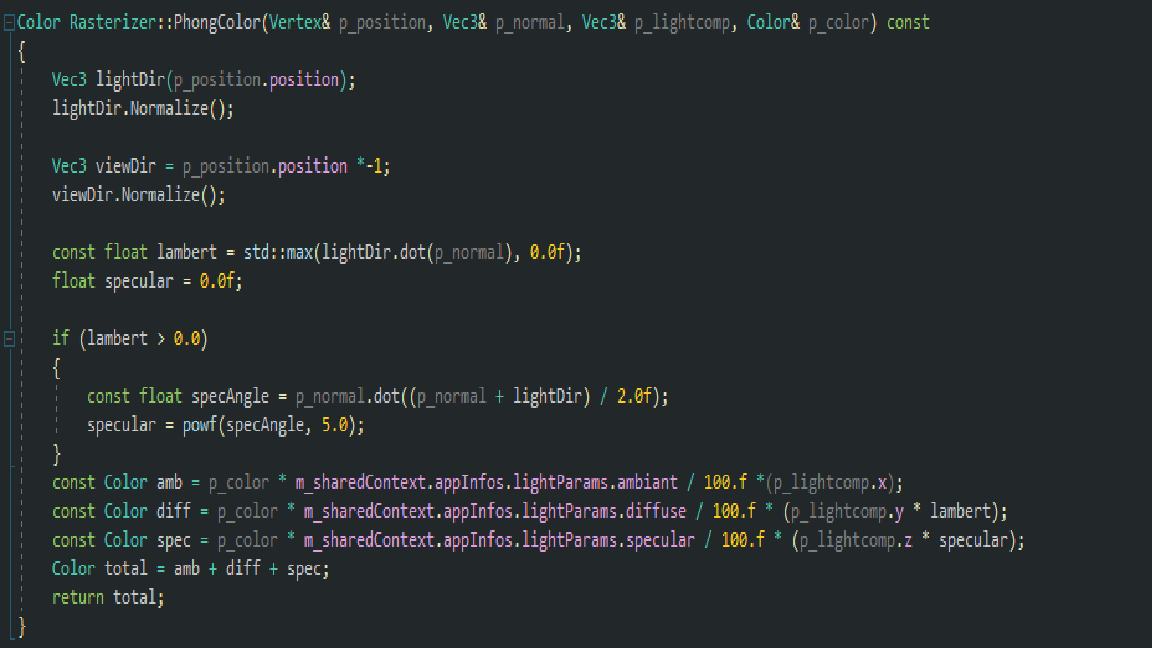
In camera space, we calculate the lighting of the object.

Because, this is the last space (in the 3D graphics) where all the coordinates are

precisely in 3D. After projection to clip space or normalized device space, the Z

coordinates will change, and the precision will not be as same as in camera

space.



* In camera space, we calculate the direction of light to the object

(since we have a direction, the calculation should use the Vector)

- We calculate the view direction, meaning the direction of the viewer

to the object.

- We calculate the angle between light direction and the normal to see

on which face exactly the light “hits” the object.

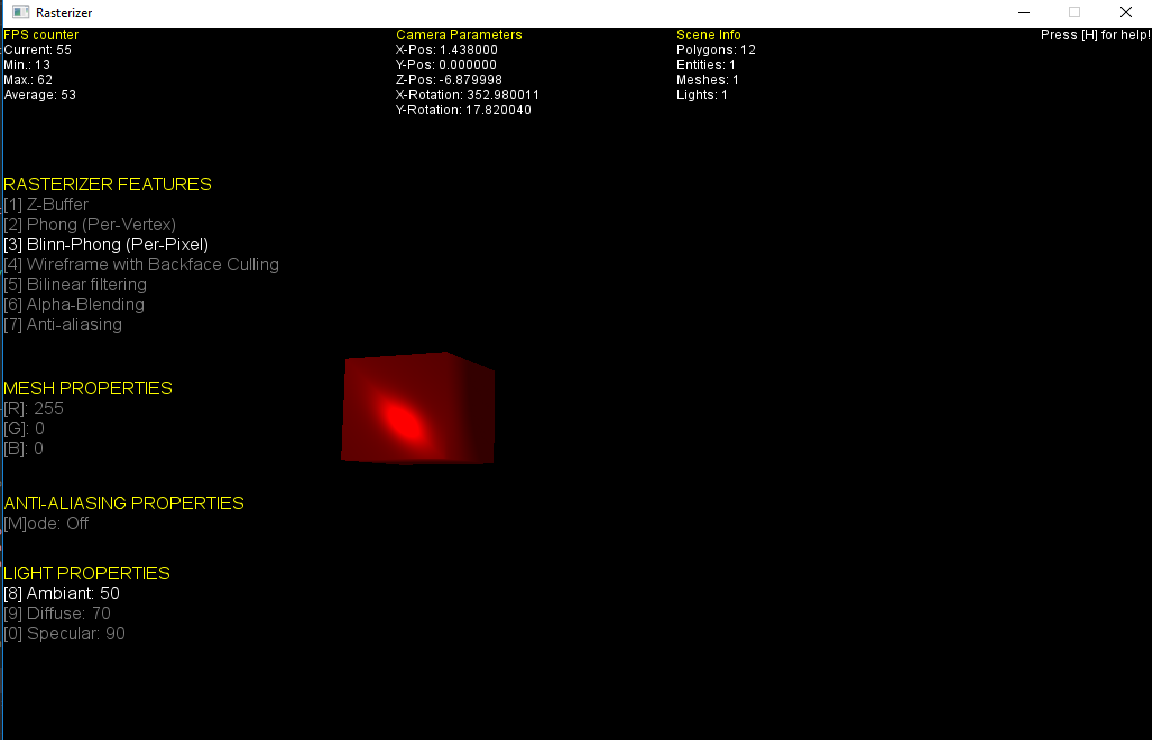
The face where the light “hits” the most will obviously be brighter than

other faces.

- We calculate the point where the light hits the most.

When the light hits a specific face, the face won’t be equally lit.

Ex. The light on point + light on a face.



After calculating the lighting, we move on to the projection space.

b.4 Camera space to the projection space

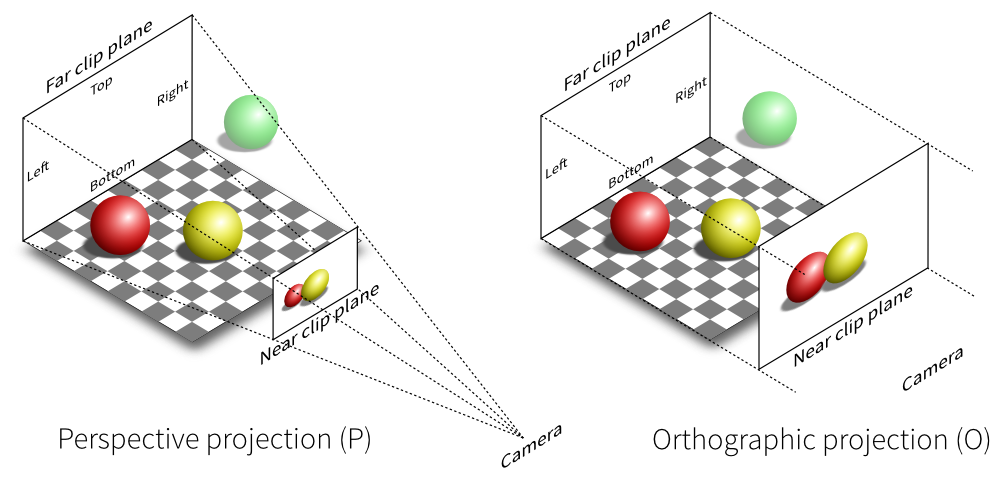
In camera space, the object’s coordinates are in 3D.

Using the projection matrix, we want to “project” these coordinates into

2D space.

We have 2 concepts of the projection.

First one is the parallel projection where putting all the objects in parallel.



The problem with this projection is the realism.

The object which is further to the view (or camera) will be drawn as big as

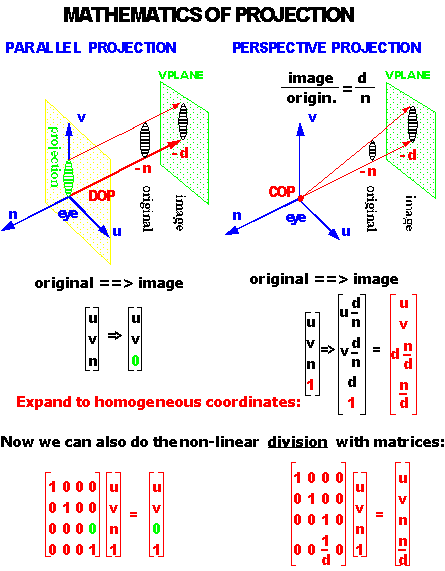
the object closer to the camera.

For the realism, we used the perspective projection.

As shown in the picture below, further the object is, it becomes smaller.



This transformation is calculated with the projection matrix.



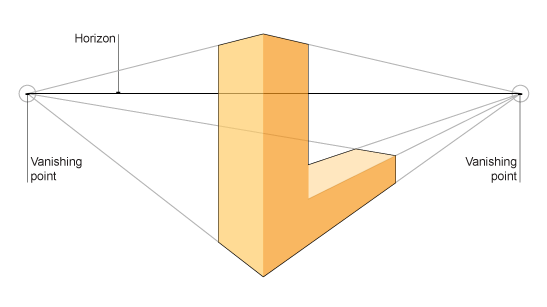
We calculate the proportion between the width and the height of the

screen and if the object is further from the screen, we make the object

smaller and always keep the proportion.

We also set the angle of view, if the angle is smaller, we will see a small

portion of the object (depending of the position).



The picture above is the example of the perspective projection.

The line closer to the camera seems bigger compared to the line further

from the camera.

b.5 projection space to normalized device space

After applying the projection matrix, we do have 2d coordinates but

These coordinates will not comply with the size of screen yet.

What we need to do is dividing all the coordinates in projection space by

4th value of the vector, which is w value.

(We will not explain in detail what the homogeneous coordinates are,

For further explanation) [3]

By dividing all the values by w, we will have normalized coordinates of x,

y and z. In other words, the values of x, y and z will be between -1 and 1.

When you have arrived at the normalized device space, there is no point

of return. The value of w will be rejected and since we are now in 2D

space. To return to the projection space, we must know the value of

w. But since it’s rejected, we cannot roll back to the projection space.

b.6 normalized device space to clip space or screen space

Clipping is not an obligation!

Clipping process is used to “cut” the positions that are out of view.

Let’s say you can only see the half of the object, what is the point of

drawing the other half? The point of rendering is to draw less.

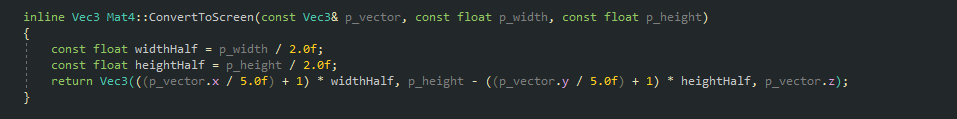
We can clip the positions using the viewport matrix, but this is not the

only way. We can do this process later in the rasterization stage.

With the normalized coordinates, we now must make them “fit” to the

given screen. In our case, the limiting coordinates of the screen are

-5 to 5 for x and y.



Using the function multiplied to each coordinate, we now have the

screen coordinates.

C. Rasterization stage

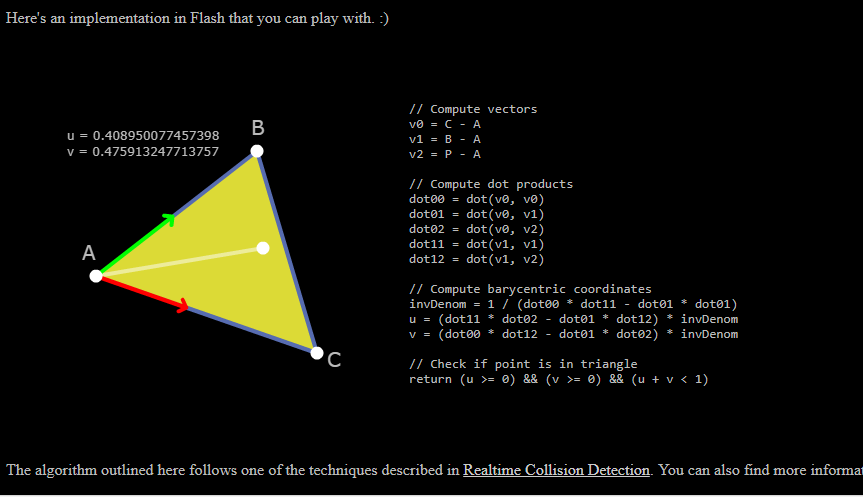
Now we have all the coordinates in the screen space.

All we do in the rasterization stage is giving colors to each pixel.

But the problem is, we need to know which color should be given to which pixel.

To do this, we use the famous barycentric technique.

Using the barycentric technique, we calculate whether a specific point is in a triangle. [4]



In the picture above. Red line represents the vector from A to C and the green line, the

vector from A to B.

the values of these vectors vary from 0 to 1 if the point is in the triangle.

If the values are less than 0 or bigger than 1. The point is not in the triangle.

If the given point is closer to the point B, green vector becomes bigger.

If the given point is closer to the point C, red vector becomes bigger.

Lastly, if the given point is closer to the point A, both vectors become smaller.

By using these vectors, we can evaluate different factors.

* Interpolate the z position for Z -buffer
* Interpolate the position for color blending
* Interpolate normal vector for lighting
* Interpolate the edge for anti-aliasing
* Etc.

Using the barycentric technique, the computation is heavy, but we can make

Every other step faster.



Interpolation of color can be much easier with barycentric technique.

C.1 Z-Buffer

Z-Buffer algorithm is used to sort what must be drawn first.

It’s called the painter’s algorithm where we draw the further object first.

Let’s say we want to draw a letter A,

Without the depth test, we will be what must draw in the back in the front and

what must be drawn in the front in the back.

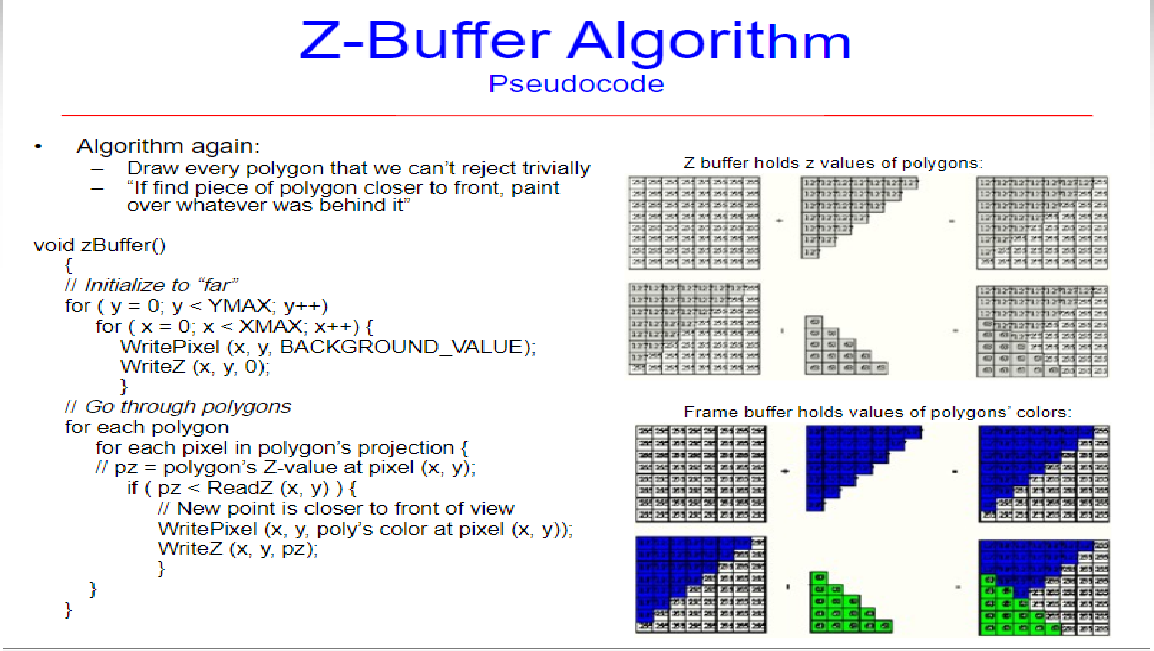


We use a float array, same size of the screen.

“zBufferArray(new float[ScreenWidth \* ScreenHeight])”

On each position of pixel, we save the z value of each pixel and sort in which

order the object must be drawn.



C.2 texturing

Texturing is putting an image file, such as .png, .bmp, .jpg, into each triangle of

vertices.

An image file has a header containing important information about the file:

The width, the height of an image, the color of each pixel, the file format, etc.

After decoding the header file, we now have the size of the image and the RBGA

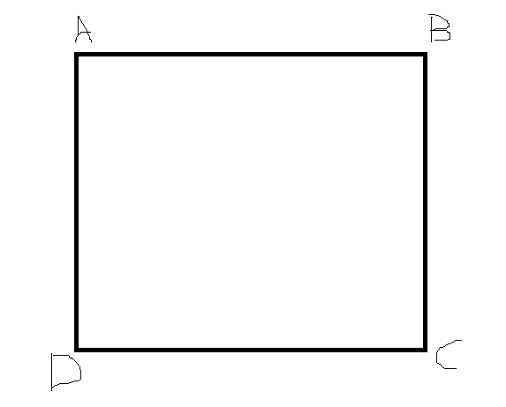
order of the image.

First, we bind each vertex coordinates with U and V coordinates for image.

U/V coordinates are normalized coordinates, its’ values can vary from 0 to 1.

1 means the full width or height of the image. 0, the starting point of the image.

Let’s say we want to put 800 X 600 .png image into a square.



A (-0.5, 0.5, 0), B (0.5, 0.5, 0), C (0.5, -0.5, 0), D (-0.5, -0.5, 0).

To put a letter S image into the square ABCE, we set upper-left corner of image

to (0, 1). 0 for width, 1 for height.

Right-left to (1, 1) 1 for width and 1 for height and so on.

After binding these coordinates, you should have

A (-0.5, 0.5, 0) and UV(0, 1)

B (0.5, 0.5, 0) and UV(1, 1)

C (0.5, -0.5, 0) and UV(1, 0)

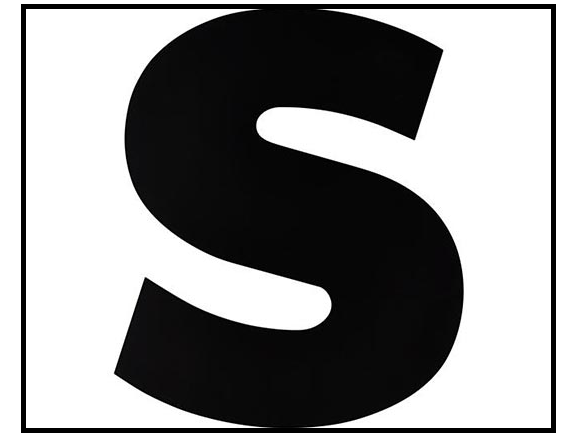
D (-0.5, -0.5, 0) and UV(0, 0)

With these coordinates, use barycentric technique to determine which pixel in

the image corresponds to the pixel on the screen.

After the calculation, we copy the RBGA color information from the pixel in the

image and paste it to the corresponding pixel on the screen.

And the result: 

C.3 Alpha Blending and back-face Culling

Alpha blending is transparency of an object.

If two objects are drawn in the same positions and if the object in front is

transparent, we must be able to see those 2 objects.

Ex. Alpha blending



Before processing the alpha blending, the back-face culling must be performed

first.

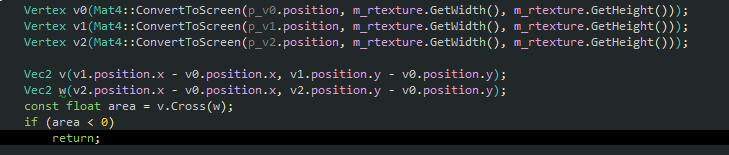
Back-face culling is “not drawing” all the triangles that are not seen from the

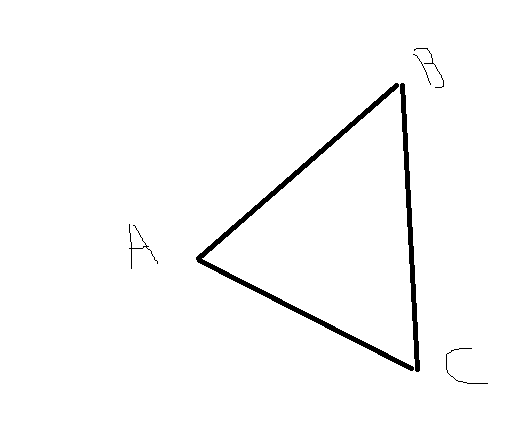
viewer.

Normal vector represents which direction the triangle (of polygon) is facing.

If Normal is negative, this means it’s facing away from the camera, so it won’t be

seen by the viewer.





Let’s say we have a triangle like this.

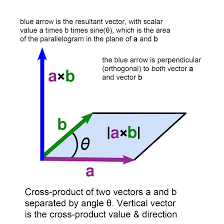
First, we calculate AB vector and AC vector.

Second, we calculate cross product of these vector.

The result is a vector that is perpendicular to both AB and AC.

So, if this vector is pointing toward the viewer, its’ value is positive, and it is

visible.



Now that we erased the “back-face” of the object, we need to perform the alpha

blending.

The algorithm of alpha blending is not drawing what is behind in front nor

making what is behind visible.

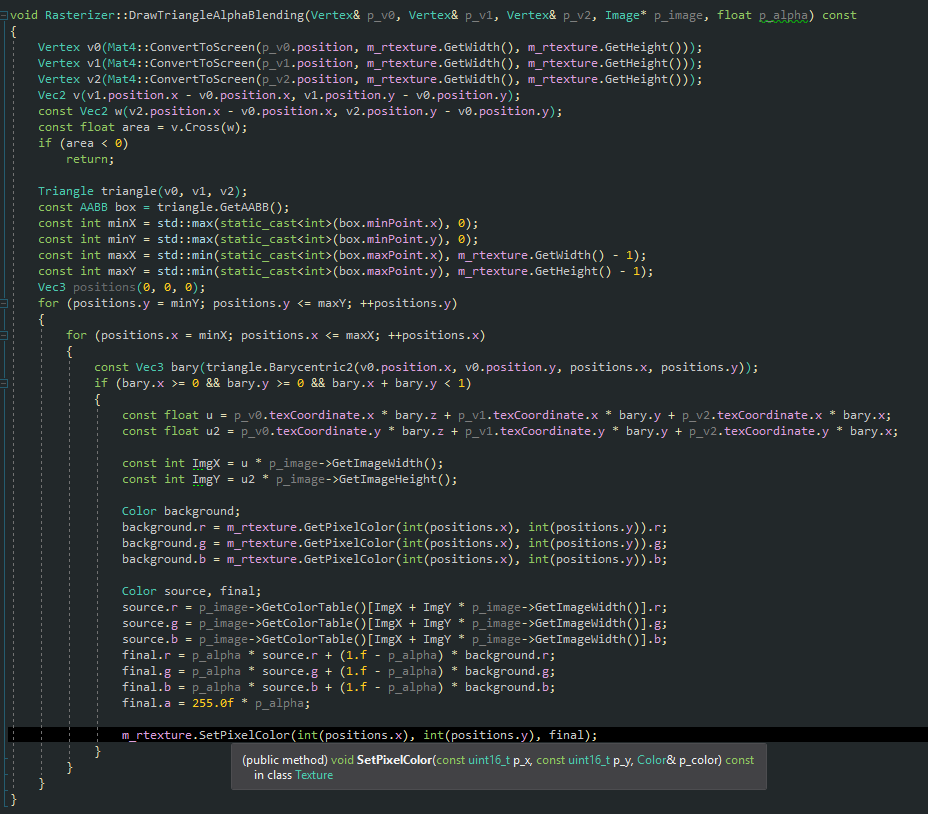
We also need to disable Z-buffer.

To perform the alpha blending, we need to calculate the color of the overlapping

objects and mix them (blending), proportion to the alpha value.

If the object in front has alpha value of 0.4, 40% of the color in front will be

mixed with 60% of the color in behind.

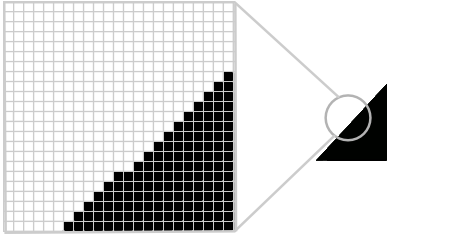


C.4 Anti-aliasing

“In [digital signal processing](https://en.wikipedia.org/wiki/Digital_signal_processing), **spatial anti-aliasing** is the technique of minimizing the distortion artifacts known as [aliasing](https://en.wikipedia.org/wiki/Aliasing) when representing a high-resolution image at a lower resolution. Anti-aliasing is used in [digital photography](https://en.wikipedia.org/wiki/Digital_photography), [computer graphics](https://en.wikipedia.org/wiki/Computer_graphics), [digital audio](https://en.wikipedia.org/wiki/Digital_audio), and many other applications.”[5]

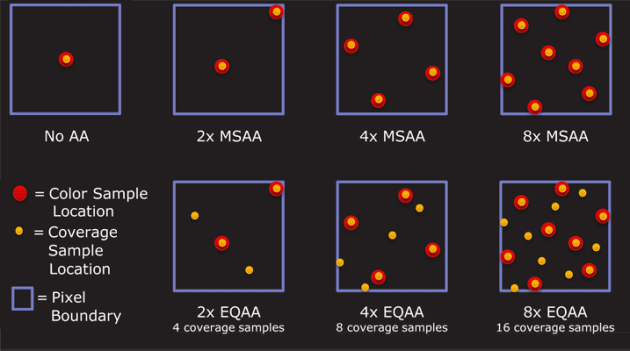
Since all pixel values are calculated in int, the line which should be calculated in

float value will be approximative.



To correct this distortion, we need to divide each pixel on the edge into sub-

pixels.



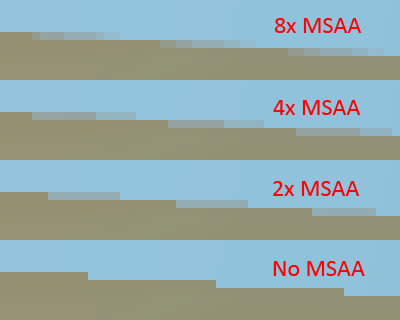
Like the picture above, we choose 2 to 16 points inside of a pixel and check

If these points are in the triangle or the edge.

With total number of the points and the number of points that are in the edge or

the triangle, we calculate the “average” color of the pixel.

More sample points you have, more precise the line becomes.



 letter C with no antialiasing.

letter C with antialiasing.

1. Missing information
2. Camera

In openGL, there is a function called gluLookat to set the camera.

In our case, since we fix the camera at the origin, we do not need to implement a

function to manipulate the camera.

We just implement a function to create an inverse matrix.

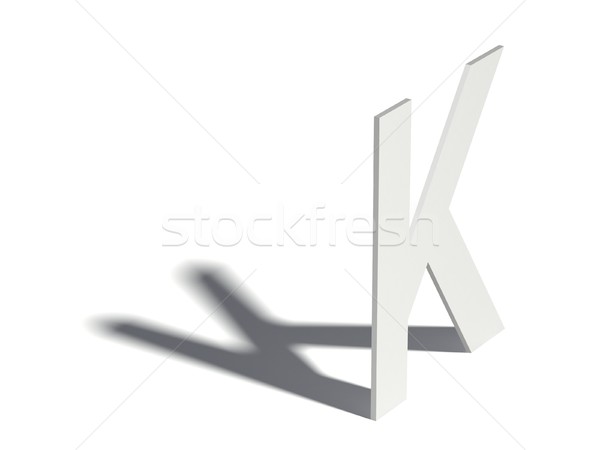
If we move the object to the left, inverse Matrix will allow us to move it to the right.

Technically it’s not a real camera but it demonstrates the use of inverse matrix.

1. Further implementation?

We can implement the shadow mapping, but this required the

knowledge in Calculus.

 (letter K with shadow).

We can also implement the reflection effect.



(Portal 2, one of the best implementation of reflection).

Lastly, we could also implement anti-aliasing on a texture.



P.S: There is a hidden code that unlocks the special feature in our project.

If you followed well our documentation, you cannot miss it.

If you found it, type it while running our project.

1. [CIE](https://en.wikipedia.org/wiki/International_Commission_on_Illumination) (1987). [*International Lighting Vocabulary*](http://www.cie.co.at/publ/abst/17-4-89.html). Number 17.4. CIE, 4th edition. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [978-3-900734-07-7](https://en.wikipedia.org/wiki/Special:BookSources/978-3-900734-07-7).  
By the *International Lighting Vocabulary*, the definition of *light* is: “Any radiation capable of causing a visual sensation directly.”

2. <https://en.wikipedia.org/wiki/Camera_obscura>

*3.* [*https://en.wikipedia.org/wiki/Homogeneous\_coordinates*](https://en.wikipedia.org/wiki/Homogeneous_coordinates)

*4.* [*http://mathworld.wolfram.com/BarycentricCoordinates.html*](http://mathworld.wolfram.com/BarycentricCoordinates.html)

*5.* [*https://en.wikipedia.org/wiki/Spatial\_anti-aliasing*](https://en.wikipedia.org/wiki/Spatial_anti-aliasing)