# 1 1. Introduction

## 1.1 History

In the 1950s and 1060s, the cost for CRTs was too high, since the computers were too expensive and slow. Mah boy Cohen Sutherland developed many of the algos responsible fro todays graphics. The idea of a **raster**, or an array of pixels, came along as a way to represent images.
In the 80s and 90s, there came more hardware to do some of the calcualtions required.

## 1.2 Image Formation

In an image, there are **objects**, **viewers**, and **light sources**. Different attributes govern how all of these elements interact in common.
We can have rays of light come out of the light source. **Light** is a part of the EM spectrum that us humans can perceive with our light. The technique called **ray-tracing** involves following an individual light source and then attempts to figure out whether it enters the camera. The issue with this approach is that a single light source can have multiple interactions before entering the lens of the camera, so figuring out if it made it into the scene will take a lot of processing power for all the light rays.

## 1.3 Color

The prevailing theory of how humans perceive color is through *three-color theory*, which states that humans have two types of receptors in our eyes that together capture. There is *additive color* and *subtractive color*, which means that

## 1.4 camera

When talking about a camera, we need several things: the *image plane*, where the image is being projected onto a plane, the *projector*, which directs the iamge from the image plane to the center of projection. With a simple **pinhole camera**, the image essentially gets flipped upside down after entering the peephole, since the lens restricts the light coming in. The perspective equations are:

$$x_p = -x(z/d)$$

$$y_p = -y/(z/d)$$

$$z_p = d$$

Remember the perspective projection uses similar triangles! Essentially, we have to pass from the original clipping plane to the plane encompassing $(1,1,1) - (-1,-1,-1)$. These formulas are how we apply the transformations.

## 1.5 image representation

Because images are hard to compute, instead we use **primitives**, which allow us to break down an arbitrarily complex figure into a collection of simpler polygons. Since triangles are the simplest polygons, the hardware is usually specially designed to deal with polygons. After we draw the polygons, we need to remove *hidden surfaces*.

# 2 Clipping

(I cover this out of order because I had not studied this).

## 2.1 Objectives

For clipping, we want to make sure we know what objects make it into the scene. Once we know what things make it into the scene, we need to know what objects are in front of others. We need to know clipping, which is easy for lines and polygons, but harder for other types of figures.

## 2.2 2d Clipping

If we were using the brute force approach, we would calculate all the intersections with the sides of the viewing rectangle.

## 2.3 Cohen-Sutherland Algorithm

Without calculating intersections, we can eliminate some of the lines. The most important thing is to calculate the window. We then have a couple cases:

- Case 1: Both endpoints of a line segment are within the window. Accept the line.
- Case 2: Both endpoints lie on the outside and on the same side of the line(they don't come in to the window). Reject the segment.
- Case 3: One endpoint inside and one outside. Here we have to perform some calculations.
- Case 4: If they are both outside, we might have to do some calculations if they have one part inside.

## 2.4 Outcodes

Because we have four pairs of intersecting lines, we need to calculate 9 outcodes, which we need 4 bits for. We will need at most four subtractions to calculate the intersection. The initial point $C$ and endpoint $D$ of a line-segment will each land on a segment. We compare the subtraction of the two endpoints to see if they both lie on the same section.
If the logical ANDing of the two outcodes does not equal 0, then we know that the segment lies outside the region and we reject it. Else if the the logical AND yields 0 and the segments are not the same, we will need to calculate the intersection.

## 2.5 efficiency

Usually the efficiency is not bad since the clipping window is usually relatively small to the size of the database.

## 2.6 Polygons

*Convex* polygons can only create one other shape when passed through this algorithm. If the polygon is non-convex, then we need to *tesselate* it, or split it into smaller triangles.

## 2.7 Pipelining

Since the intersections on the sides can be calculated independently, we can insted create a separate pipeline accross the sides of the window. The *Sutherland-Hodgman* algorithm can detect the overlapping segments of a non-convex polygon by extending the sides of the clipping polygons indefinitely. The Cohen-Sutherland algorithm can be used in 3d, however it requires 6-bit outcodes instead of 4.

# 3 Programmable Shaders

Essentially, since the first development of GPUs, after a while we were able to develop programmable shaders. This means we can customize per-vertex functions such as Phong shading, colors, normal transformation, etc...
What is a *fragment*? "potential pixel". We can perform per-fragment calculations without knowledge about other fragments.

## 3.1 D3D 10 Pipeline

- Input assembler: Receives the sets of coordinates to the pipeline.

- Vertex Shader: does vertices, transformation, skinning, lighting

...etc

# 4 Shading

## 4.1 Illumination

We have different sources of light/shading, and different surface properties.

## 4.2 Light-Matieral Interaction

Reflection of light depends the surface of the object. The *rendering equation* works over all the space, since we have see how the light reflections affect a specific region, however this is impractical for real-time usage. **Local Illumination** refers to the lighting of just the specific vertices that

## 4.3 Light Sources

Light sources are complicated b/c they emit light for more than one point. Then, more than one light ray from a source would reach the source, and we would have to integrate over the entire sruface.

## 4.4 Light Sources

- Point source:There is just a point that emits color in all directions. Infinite distance lights would mean the light rays coming to the scene are parallel.
- Spotlight:There is a cone which emits light in a restricted area.
- Ambient Light: Ambient light would generate equal lighting all over the scene.

The smoother a surface, the more the light will be reflected off of its surface. Because a rough surface will reflect rays more erratically, the light will reflect in all directions.

## 4.5 Phong Reflection Model

The reflection model used by Phong keeps four vectors: the vector to the source $l$, the vector to the viewer $v$, the normal $n$, and the vector assuming perfect reflection $r$.
The three components of Phong Lighting are: Ambient, Diffuse, Specular. *Diffuse* light scatters light in all directions. The intensity of the light will depend on the angle of reflection, since it would be spread over a larger area.
*Specular* surfaces reflect light close to the region of a perfect reflection. $l_r k_s l \cos^a \phi$, which relates reflected intensity, absorption coeff, incoming intensity, and shininess coefficient. The shininess coefficient can change the texture of a material. Higher values correspond to more metallic looks.
We also have to add a term for the distance from a source, which is inversely proportional to the distance.
The modified Phong Model addresses the problem of using the complicated complication, so instead we use the *halfway angle*.

## 4.6 Computation of Vectors

$I$ and $v$, which are specified by the application. $r$, which is computed from $l$ and $n$.

## 4.7 Shading

How do we fill color within a polygon? Flat shading, Gouraud shading, or Phong Shading.
*Flat Shading* assumes that the vectors are constant.
*Gouraud Shading* finds the average noraml at each vertex. However, this can lead to *Mach Bands*, becuase the contrast between the planes is still sharp. Take the normal of the vertex and find the value corresponding it, and apply it to the planes.
*Phong Shading* finds the vertex normal, and interpolates vertex normals, and we find shades along the edges.