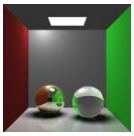


2018/2019, 4th quarter

INFOGR: Graphics / *NeedToKnow*

Lecture 5 – Rasters - Colors - Ray Tracing

Author: Jacco Bikker

TL;DR

In this installment of 'NeedToKnow': rasters: which we use when discretizing a continuous virtual world to a finite set of rectangular pixels; colors: in nature simply a collection of wavelengths in the visible light spectrum, but on a computer again discretized and bound; and ray tracing: one of the two main algorithms for producing pretty images in this course.

As promised, this document details everything you need to know. On top of that plenty of (pointers to) additional material is provided. Everything outside the main text, including links, is optional reading, unless explicitly specified.

Rasters

Data in the real world is typically analog*. We can always pick a location between two locations, we can measure a sound with arbitrary accuracy, temperatures and speeds are continuous variables. When working with a computer we need to turn to digital data. Even a float or double in C++ or C# is digital data: in the end there are (close to) 2^{32} unique float values, which means that there is a minimum distance between two values. Regularly spaced values are called *discrete*, and a signal that consists of such values is a *discrete signal*. The conversion of an *analog signal* to a *digital signal* is called *discretization*. We need this for storing audio on a CD: the audio signal is sampled 44.1k times per second, and the probed values are stored using 16 bits. That means: 65536 unique amplitudes.

For a display we do something similar. We have a finite number of pixels, e.g. 1920x1080, on which we often want to display continuous shapes, such as lines or circles. Converting a 2D signal to a raster is a form of discretization, which we can also call *rasterization*.

The process of rasterization can be performed at various levels of fidelity. Perfect rasterization is computationally expensive: each pixel should have the average color of everything we can see through it. The pixel color is the *integral* over all possible view directions through the rectangle of the pixel, or the average color of an infinite number of random samples over the area of the pixel.

In the lecture we discussed three ways to improve the quality of rasterization:

1. increase resolution, so we get smaller pixels;
2. anti-aliasing, which approximates the integral;
3. animation, which relies on our eyes to 'connect the dots'.

That's just my random selection; feel free to discuss additional ways on Slack.

---

* Well… let's assume so. http://www.edge.org/conversation/freeman_dyson-is-life-analog-or-digital . Optional read obviously.

Raster Legacy

CRT (Cathode-ray Tube) displays have been around for a long time. They operate by firing electrons at a phosphorescent screen. The electron beam is steered horizontally and vertically using strong electromagnets.

> **DON'T TRY THIS AT HOME** – The high voltage used in CRTs is great for experiments – for example, for building your own lifter. It's an ion-propelled device that floats, and you can build it at home. I don't advice it, obviously. https://hackaday.com/2016/07/13/expanding-horizons-with-the-ion-propelled-lifter

The beam starts at the top-left corner of the display, proceeds to the top-right corner, then draws the next line, until it reaches the bottom. Finally, it returns to the top of the screen. During this *vertical retrace* the beam is turned off and no information is written to the screen.

Figure 1: Cathode-ray tube.

Early home computers had to provide the CRT with data in an efficient manner. It is thus only logical that the layout of this data follows the path of the electron beam. This left us with a coordinate system that starts at (0,0) in the top-left corner and uses integer pixel coordinates.

The time between two vertical retraces is the *frame time*. We can use it to calculate the number of *frames per second* (fps, in Hertz): 25 for an old European PAL television set; almost 30 for an American NTSC set. These days 60Hz is a common frequency, although much faster monitors exist. Frame rate is again a hardware-centric concept. The human eye is *frameless*; although there is a minimum to the length of an event that we can detect, the time at which this event starts is arbitrary.

> **LEGACY** – A CRT that displays the same image for a long period of time burns this image into the screen. To prevent this the screensaver was invented. More recent devices do not suffer from this effect, so the screensaver is just an artifact now.

Some games produce more frames per second than the monitor can display. This is actually useful: a frame can only be rendered based on data that is already prepared, which means that 100fps yields a delay of at least 10ms. Worst case is 20ms: if something happens just after the vertical retrace, a frame based on old data is rendered first (in 10ms), and only then the frame based on the new data is produced. Humans will notice such a delay, especially when using VR equipment.

Colors

It's not just geometry that needs to be digitized for display; colors are continuous in nature too and require discretization to be stored in a computer.

The most common representation for colors is 'integer RGB': each color is a mix of red, green and blue light. Setting red, green and blue to zero yields black, setting them all to the maximum value yields white. The 'maximum value' is 255: this is the largest binary value we can store in 8 bits.

Computers do not work efficiently with groups of three bytes, so we use 32-bit colors: the fourth byte is often called 'alpha' and can be used for transparency, but when displaying an image on a monitor, the value of the fourth byte is really irrelevant.

> **EXPLOIT** – The alpha data is there, whether you use it or not. Feel free to use it for a collision map in your game, or to store other data. An interesting application is integer HDR colors: the 'alpha' component stores an exponent for red, green and blue, allowing for much larger values. Something similar is done with the `GL_RGB9_E5` format in OpenGL.

If we are low on memory (or data transfer times are a concern) we can also encode our colors more efficiently using 16-bits. In that case we use only 5 bits for red and blue, and 6 for green. The extra bit is assigned to green, because that happens to be the component that our eyes are most sensitive to. We now don't have space for 'alpha', but in exchange we halved our storage.

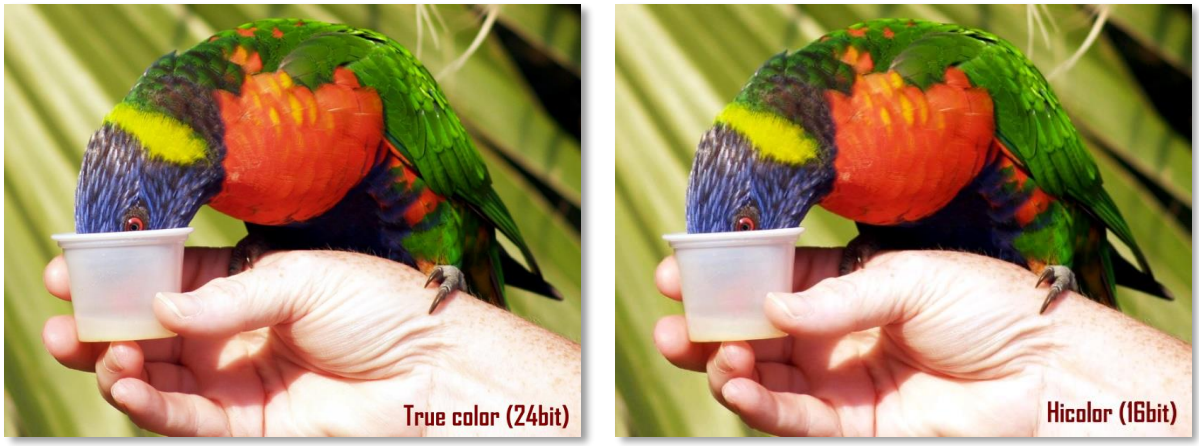


Figure 2: the difference between 24-bit and 16-bit is often hard to spot. Look for smooth gradients; 16-bit has fewer colors for the transition.

We can go even lower: by using a palette. Many images can perfectly be represented by a limited number of colors, and if this 'limited number' is 255 or less, each pixel can be represented by a single byte. The image now becomes a 'paint by numbers' image, where the pixel value is merely the index of the color, which we then need to look up in the palette. Apart from storage benefits this also can be useful for color animations, such as color cycling and

fades: we only need to fade 256 colors to change the color of the entire image, regardless of its size.

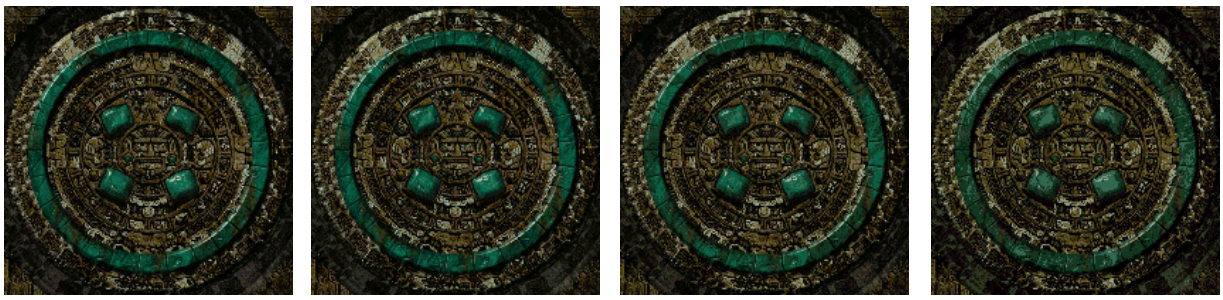


Figure 3: a texture reduced to 128, 64, 32 and 16 colors.

Integer colors have several limitations. The differences between two levels of e.g. blue are small enough for most images, but this is may no longer be the case when we start editing the image. The fact that a color has a maximum brightness is more severe: we cannot distinguish between a piece of paper and a white cloud backlit by a summer sun.

We get far more accuracy and range in our color representations if we store red, green and blue as floating point values. This requires a 32-bit number per component, but at least now we can accurately store our colors, including very bright colors. We refer to this as HDR: *High Dynamic Range*.

The 0..255 range we used for integer colors is typically mapped to 0..1 when using HDR. '255' was simply '100% of the available range' and thus as a value quite arbitrary. In floats, white becomes { 1.0, 1.0, 1.0 }. But, as said, we can go beyond that, and our bright cloud could be e.g. { 160.0, 160.0, 160.0 }.

HDR is particularly important for ray tracing. Image we have a dark bowling ball, which is quite reflective, just very dark. If we look at the reflection in this ball of a piece of paper and the bright cloud the difference between { 1, 1, 1 } and { 160, 160, 160 } is immediately clear.

Rendering

According to Wikipedia, rendering is the process of "generating a 2D image from a 2D or 3D model by means of a computer program". In this course we will investigate the two main algorithms for this: rasterization and ray tracing.

Ray Tracing

Tracing a ray by itself does not produce images, although it is a useful operation. Imagine that you have a virtual world, consisting of millions of triangles, such as a typical game level. Having the capability to find the intersection between a line which starts at a specific location and extends towards a specific direction is incredibly useful. You can use it to detect where a

bullet should cause an impact, or to check a line of sight between the player and an enemy, or to detect collisions.

But of course we can also use the rays for rendering. To do this, we create rays between the eye (or the camera) and the pixels on the screen. The rays then travel further and tell us what the first object is that we see through the pixel.

We can take one additional conceptual step: the ray becomes part of the path of a virtual photon. Photons (well at least the ones that we care about) travel between lights and the camera. The direction of the path is irrelevant; the decisions along the path remain the same. We can now start applying the laws of optics to steer the virtual photons. A photon starts at the camera, and travels though a pixel. It then hits the first object. If this object is a mirror, we calculate a new direction for the photon and proceed. If the object is not a mirror, we check visibility between each light source and the photon. If there is an unoccluded line between a light and the photon we have established light transport: from the light to the photon, and via the photon path (potentially including specular bounces) back to the camera. This algorithm is known as Whitted-style ray tracing. It is a powerful method to produce accurate images of scenes, complete with lights, shadows, glass and mirrors.

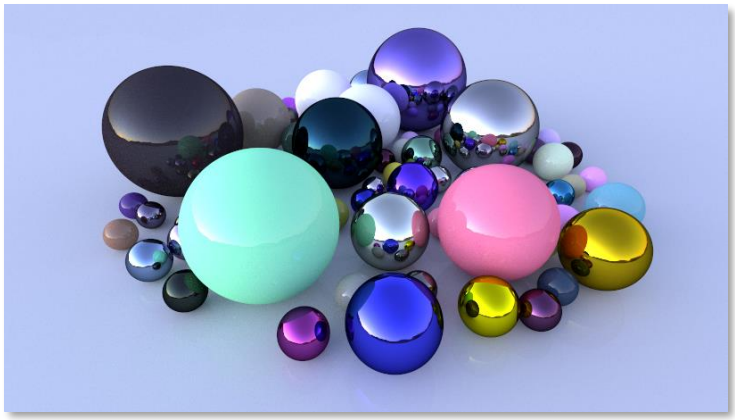


Figure 4: simple ray traced scene.

Ingredients

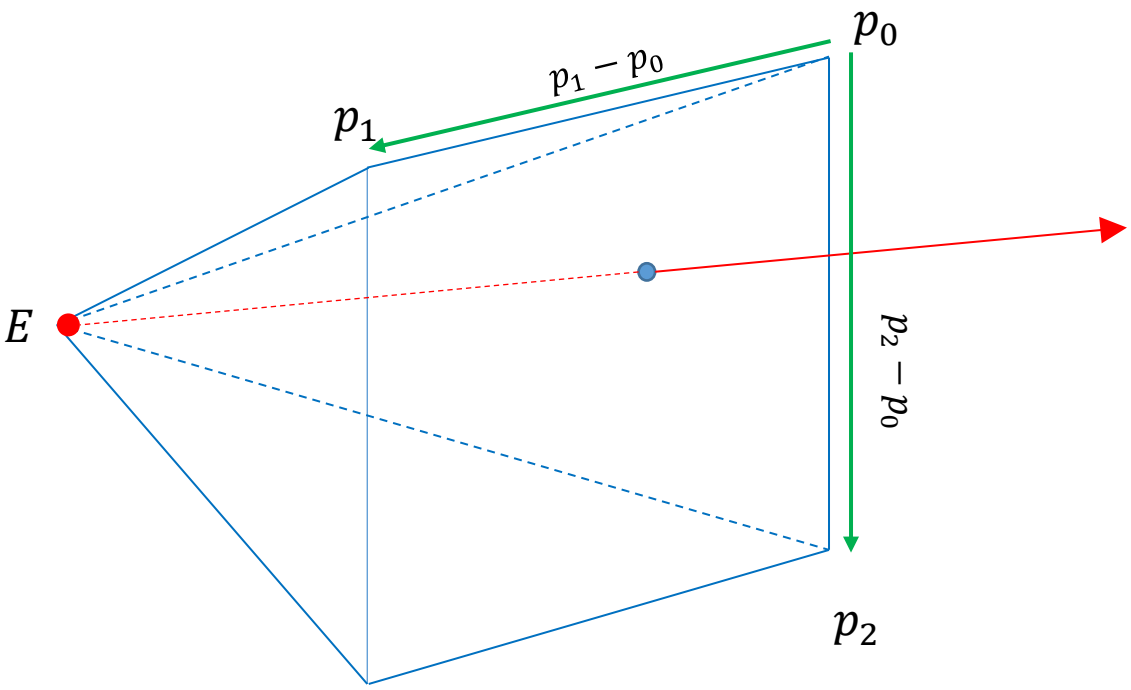
The core ingredient for ray tracing is obviously the ray. This is an 'infinite half line': it has an origin, but from there it extends into infinity. The ray is typically expressed using a parametric line equation:

$$P(t) = O + t\vec{D}, \text{ where } t \geq 0.$$

In words: a ray is the collection of points P(t) which we obtain by adding vector $\vec{D}$ t times to position O. The parameter t must be greater or equal to 0, otherwise the ray is just a line: bullets would hit the shooter and camera rays might find objects behind the player.

It is a good habit to normalize the direction of the ray. If $\vec{D}$ is normalized, t becomes a distance along the ray, which is intuitive and convenient.

We now need to aim the rays at the pixels. A direction towards a pixel is simply the position of that pixel minus the origin of the ray. The position of a pixel is obtained by interpolation over the screen plane, which is defined by its three corners (the fourth does not add information).



Using the theory from the math lectures we can see that any point on the screen is the sum of the position of the top-left corner p_0, part of the horizontal vector $p_1 - p_0$ and part of the vertical vector $p_2 - p_0$:

$$p(u, v) = p_0 + u(p_1 - p_0) + v(p_2 - p_0)$$

We now have all the information we need to setup the ray. The origin of the ray is simply the camera position. The direction of the ray is $p(u, v) - O$ and needs to be normalized. Parameter t can be used to specify a distance along the ray. Since we will be looking for the nearest object along the ray, it's best to set it to a large value. A good choice is 1e34f (i.e., $1.0 * 10^{34}$), which is close to infinity for our purposes and also close to what a float can store.

With the ray we start looking for the nearest intersection. In practice this means we intersect the ray with each primitive (plane, sphere, triangle, …), keeping the nearest intersection. The intersection itself is a purely mathematical operation.

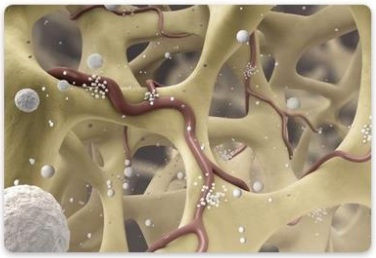
THE END

That covers the basics:

- discretization: bringing back analog values to numbers we can store in bits;
- rasters: regular collections of pixels;
- rasterization: translating shapes and scenes to pixels on a raster;
- raster legacy: a bit of history on CRTs, vertical retraces and pixel coordinates;
- discrete colors: integer, 16-bit, palettized and floating point;
- the concept of ray tracing.

In subsequent lectures we will dive deeper in the topic of ray tracing.





INFOGR2019 – NEEDTOKNOW