Maths for CG

Lecture 1 Basic Numbers and Image Manipulation

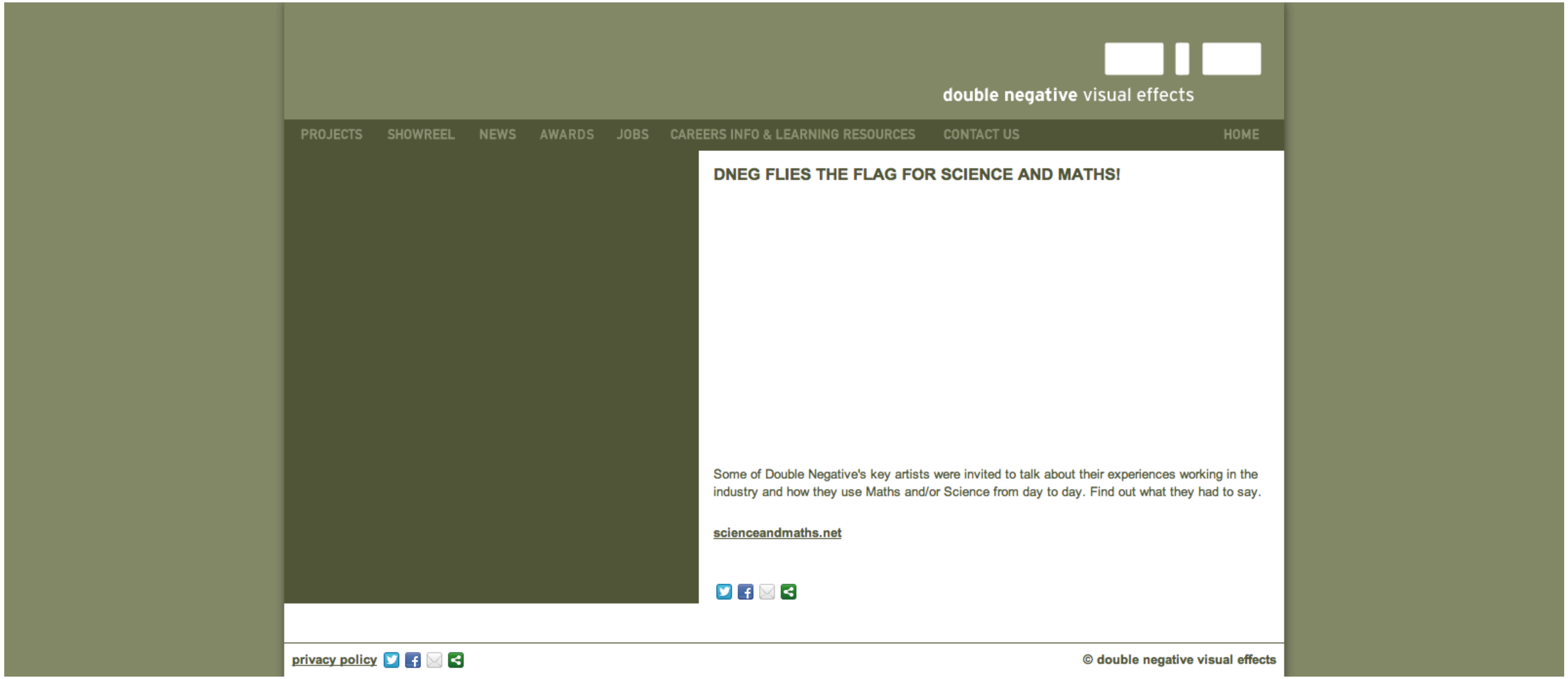
http://nccastaff.bournemouth.ac.uk/jmacey/MathsForCG/index.html

Basic Curriculum

- Basic Maths
  - Trigonometry (lines, triangles, circles)
- 3D Maths
  - Vectors, Matrices, Affine Transforms
- How Animation Systems work
  - The Computer Image, The Animation Pipeline (how XSI, Maya, Houdini, Shake etc really work)
  - Some Basic Computer Principles, How the computer Actually works
- Rendering
  - How a renderer actually produces and image, Lighting Models, Shading Models.
- And a whole lot more besides.

Why do I need Maths?

- What this video featuring some of our ex students


double negative visual effects
PROJECTS
SHOWREEL
NEWS
AWARDS
JOBS
CAREERS INFO & LEARNING RESOURCES
CONTACT US
HOME

DNEG FLIES THE FLAG FOR SCIENCE AND MATHS!

Some of Double Negative's key artists were invited to talk about their experiences working in the industry and how they use Maths and/or Science from day to day. Find out what they had to say.

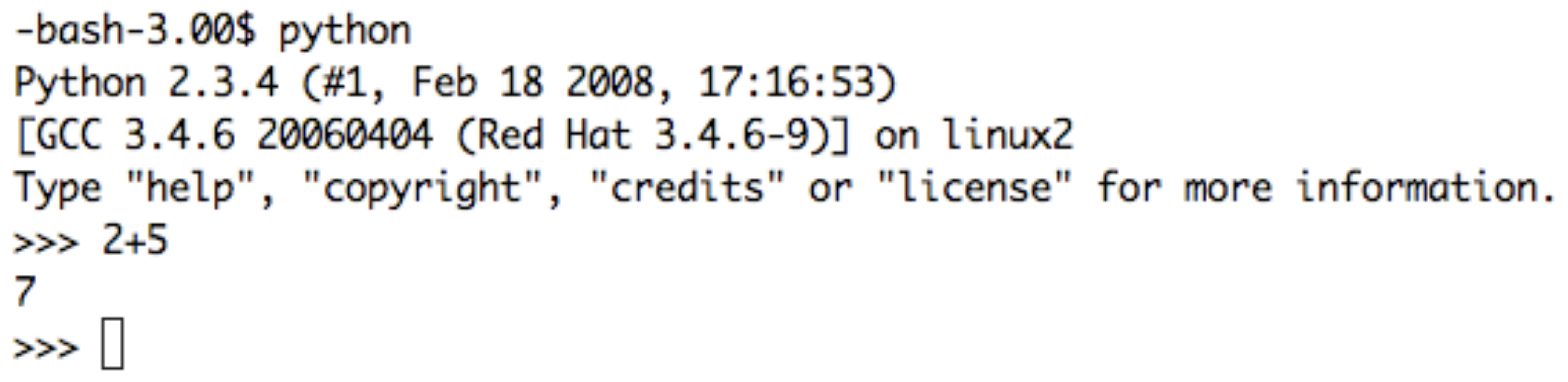
scienceandmaths.net

privacy policy

© double negative visual effects

Using Python as a Calculator

- To demonstrate some of the principles of Maths for CG we will be using the python console as a calculator
- This will allow you to get a feel for how computers process numbers and is advantageous when beginning to create scripts.



```
-bash-3.00$ python
Python 2.3.4 (#1, Feb 18 2008, 17:16:53)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+5
7
>>> 
```

Numbers

- A number is an abstract entity used originally to describe quantity.
  - i.e. 80 Students etc
  - The most familiar numbers are the natural numbers {0, 1, 2, ...} or {1, 2, 3, ...}, used for counting, and denoted by **N** or $\mathbb{N}$
  - This set is infinite but countable by definition.
  - To be unambiguous about whether zero is included or not, sometimes an index "0" is added in the former case, and a superscript "*" is added in the latter case:

$$N_o = \{0, 1, 2, ...\} N^* = \{1, 2, ...\}$$

Integers

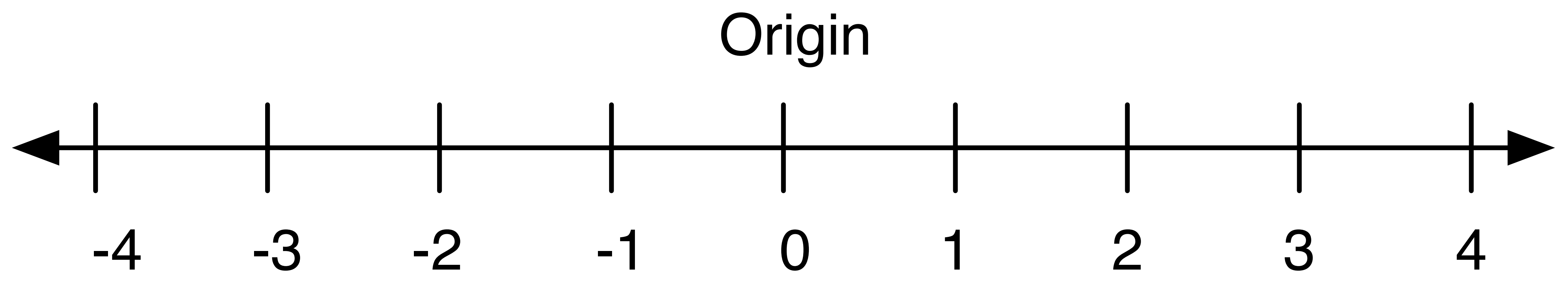
- The integers consist of the positive natural numbers (1, 2, 3, -), their negatives (-1, -2, -3, ...) and the number zero.
- The set of all integers is usually denoted in mathematics by Z or , $\mathbb{Z}$ which stands for Zahlen (German for "numbers").
- They are also known as the whole numbers, although that term is also used to refer only to the positive integers (with or without zero).

Integers in Computers

- Most programming languages have an integer data type
- They are usually used for counting and index values

```
An integer is usually in the range -32,768 to 32,767
    .

//C++ / C
int a=1;

// Maya Mel
int $a=1

// Python
a=1

```

Number Lines



- The number line is a diagram that helps visualise numbers and their relationships to each other.

- The numbers corresponding to the points on the line are called the **co-ordinates** of the points.

- The distance between to consecutive integers is called a **unit** and is the same for any two consecutive integers.

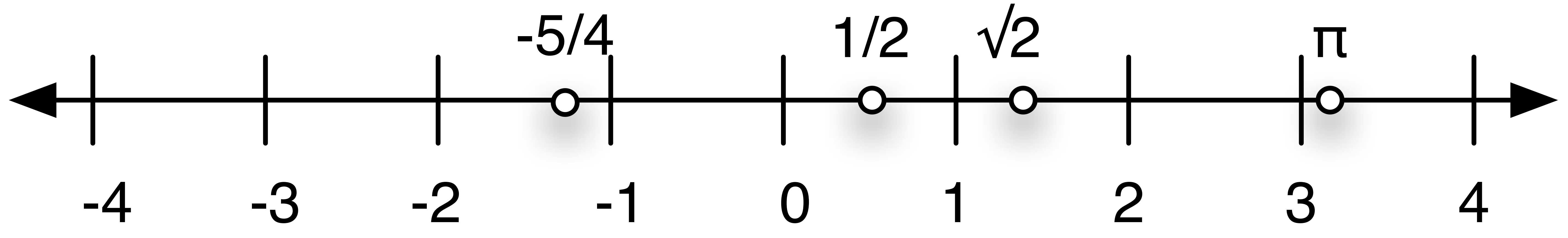
- The point with co-ordinate 0 is called the origin.

Rational Numbers

- In mathematics, a rational number (or informally fraction) is a ratio or quotient of two integers
- It is usually written as the vulgar fraction $\frac{a}{b}$ or a/b, where b is not zero.
- Each rational number can be written in infinitely many forms, for example

$$\frac{3}{6} = \frac{2}{4} = \frac{1}{2}$$

Real Numbers



- For every rational number there is a point on the number line
- For example the number $\frac{1}{2}$ corresponds to a point halfway between 0 and 1 on the number line.
- and $-\frac{5}{4}$ corresponds to a point one and one quarter units to the left of 0.
- Since there is a correspondence between the numbers and points on the number line the points are often referred to as numbers
- The set of points that corresponds to all points on a number line are called the set of Real Numbers.

Real Numbers

- Real numbers may be rational or irrational; algebraic or transcendental; and positive, negative, or zero.
- Real numbers measure continuous quantities.
- They may in theory be expressed by decimal fractions that have an infinite sequence of digits to the right of the decimal point;
- Measurements in the physical sciences are almost always conceived as approximations to real numbers.

Real Numbers

- Computers can only approximate most real numbers with rational numbers; these approximations are known as floating point numbers or fixed-point numbers.
- Computer algebra systems are able to treat some real numbers exactly by storing an algebraic description (such as "sqrt(2)") rather than their decimal approximation.
- Mathematicians use the symbol R or alternatively $\mathbb{R}$ to represent these numbers

floating point data types

- Real numbers are represented using floating point data types.
- Most computer systems have several ways of representing them
- The different representations are dependent upon the number of bit the operating system use (32 vs 64)
- In most languages we have two sizes for our real data types.

real data types

```
// A float is usually in the range +/- 3.4 e +/- 30 (~7 Digits)

// C++ / C
float a=2.5;

// Mel
float $a=2.5;

// Python

a=2.5
```

Symbols

- Mathematicians use all sorts of symbols to substitute for natural language expressions.
- Here are some examples

$<$	less than
$>$	greater than
$\leq$	less than or equal to
$\geq$	greater than or equal to
$\approx$	approximately equal
$\equiv$	equivalent to
$\neq$	not equal to

```
# less than
a<b
# greater than
a>b
# less than or equal to
a<=b
# greater than or equal to
a>=b
# Not equal
a!=b
# equality
a==b
```

Boolean Values

- Boolean values are used to hold truth values.
- Typically in a computer these values are stored as
  - 1 == true
  - 0 == false
- Some modern languages actually use the values true and false to make the code more readable.
- We use boolean values to store the results of comparisons

```
#!/usr/bin/python

a=1
b=1

print a<b
print a>b
print a<=b
print a>=b
print a!=b
print a==b
```

```
False
False
True
True
False
True
```

Comparing Floating point values

- floating point data values are not accurate and comparing them may lead to strange problems (especially in Python)
- Consider the following python example

```python
#!/usr/bin/python
# declare a to be equal to 0.2+0.1 (should be 0.3)
a=(0.2+0.1)
# declare b as 0.3 directly
b=0.3


print "testing␣equality"

if( a == b) :
  print a,b,"␣are␣the␣same"
else :
  print a,b,"␣are␣different"
```

Better Comparison

- Usually with floating point values we compare to see if we are close
- To do this we use an error margin (usually called epsilon)

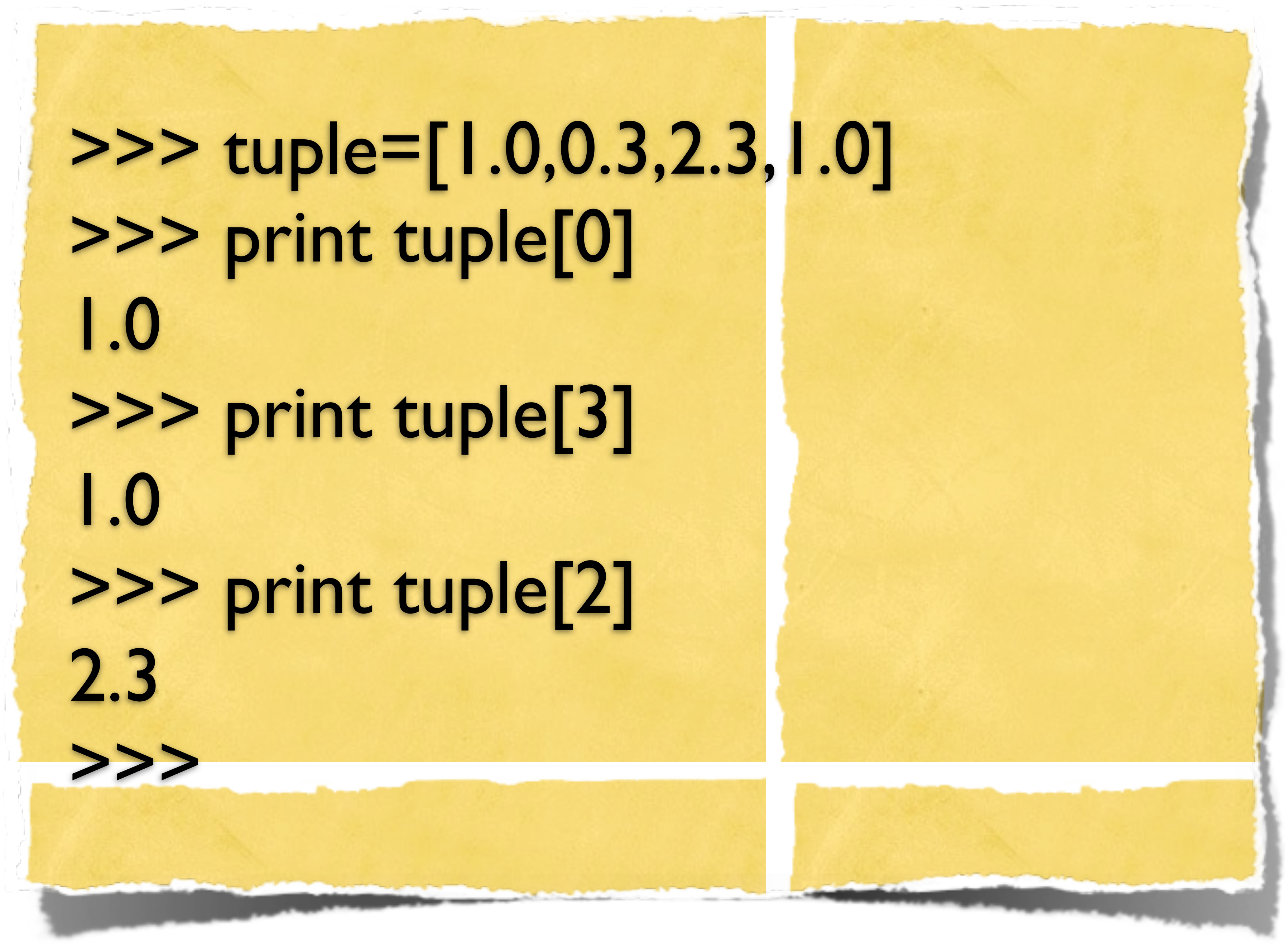
```
#!/usr/bin/python


def FCompare(a,b) :
  epsilon = 0.0000000000001
  return ( ((a)-epsilon)<(b) and ((a)+epsilon)>(b) )

# declare a to be equal to 0.2+0.1 (should be 0.3)
a=(0.2+0.1)
# declare b as 0.3 directly
b=0.3


print "testing␣equality"

if( FCompare(a,b)) :
  print a,b,"␣are␣the␣same"
else :
  print a,b,"␣are␣different"
```

Tuples

- In mathematics and computer science, a tuple is an ordered list of elements.
- We use tuples to represent a number of things in computer graphics
- For example
- Colour
  - [1.0,0.0,0.0,1.0] R,G,B,A
- Position
  - [2.5,3.4,2.0,1.0] X,Y,Z,W

Operations on Tuples

- A tuple provides a simple way of combining related data together in an ordered set
- We can access the individual elements using an integer index as follows



```
>>> tuple=[1.0,0.3,2.3,1.0]
>>> print tuple[0]
1.0
>>> print tuple[3]
1.0
>>> print tuple[2]
2.3
>>>
```

Piecewise Operations

- It is not uncommon to execute piecewise operations on tuple data
- For example we may wish to scale all the elements
- Or add two tuples together

[0.5,1.0,0.2,1.0]*2 = [1.0,2.0,0.4,2.0]
[0.5,1.0,0.2,1.0]+[0.3,1.0,0.8,2.0] =
[0.8,2.0,1.0,3.0]

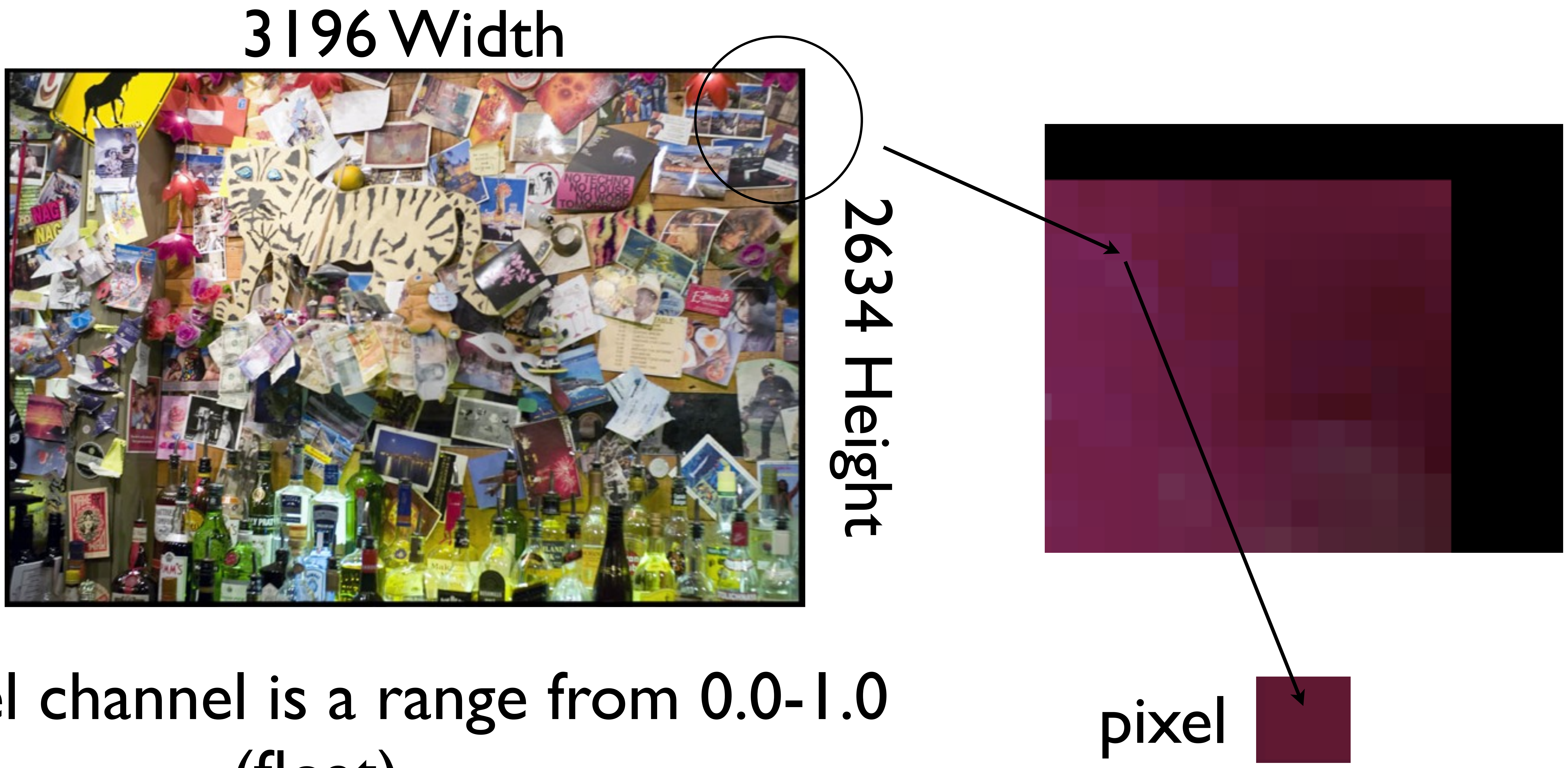
Representing Colour

- We can represent colour using many different methods, however the most common in CGI is the RGB or RGBA tuple
- Typically we represent colour as intensity values where
  - 0.0 is no intensity
  - 1.0 is full intensity
- We then combine these together into a tuple to represent colour

[1.0,0.0,0.0] = Red channel full intensity
[1.0,1.0,1.0,1.0] = White with full opacity

2D Images

- Typical 2D images are a collection of tuples where each tuple represents a single pixel
- This simple bitmapped format is the easiest to store and manipulate
- There is a 1-1 relationship between the pixel value and the data stored
- There is no attempt at image compression or saving space



Each Pixel channel is a range from 0.0-1.0 (float)

Each pixel has 3 channels (RGB)

Therefore we have

(3196*2634)*3*sizeof(float) bytes

101019168 bytes of data

0.37	0.003	0.198
🟥	⬛	🟦
Red	Green	Blue

Piecewise Image Operations

- In the previous example we can see that the image is represented in a 2 dimensional grid or table
- We can access the elements using integer cartesian co-ordinates to grab a pixel at the row and column
- Typically we represent the top left corner as having a co-ordinate or 0,0
- The bottom left as [width,height]
- We can write an algorithm to apply some sort of operation on a per pixel basis

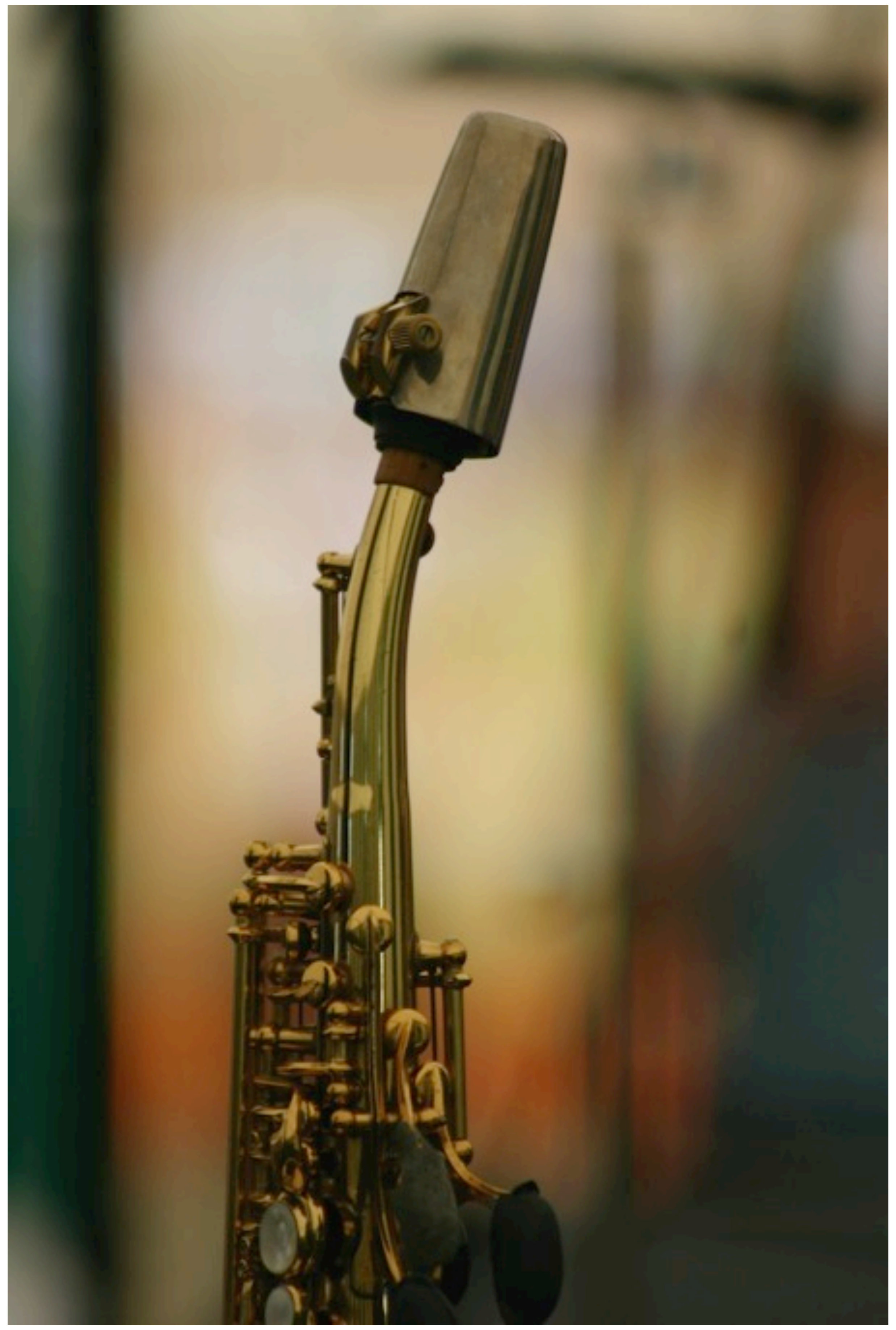
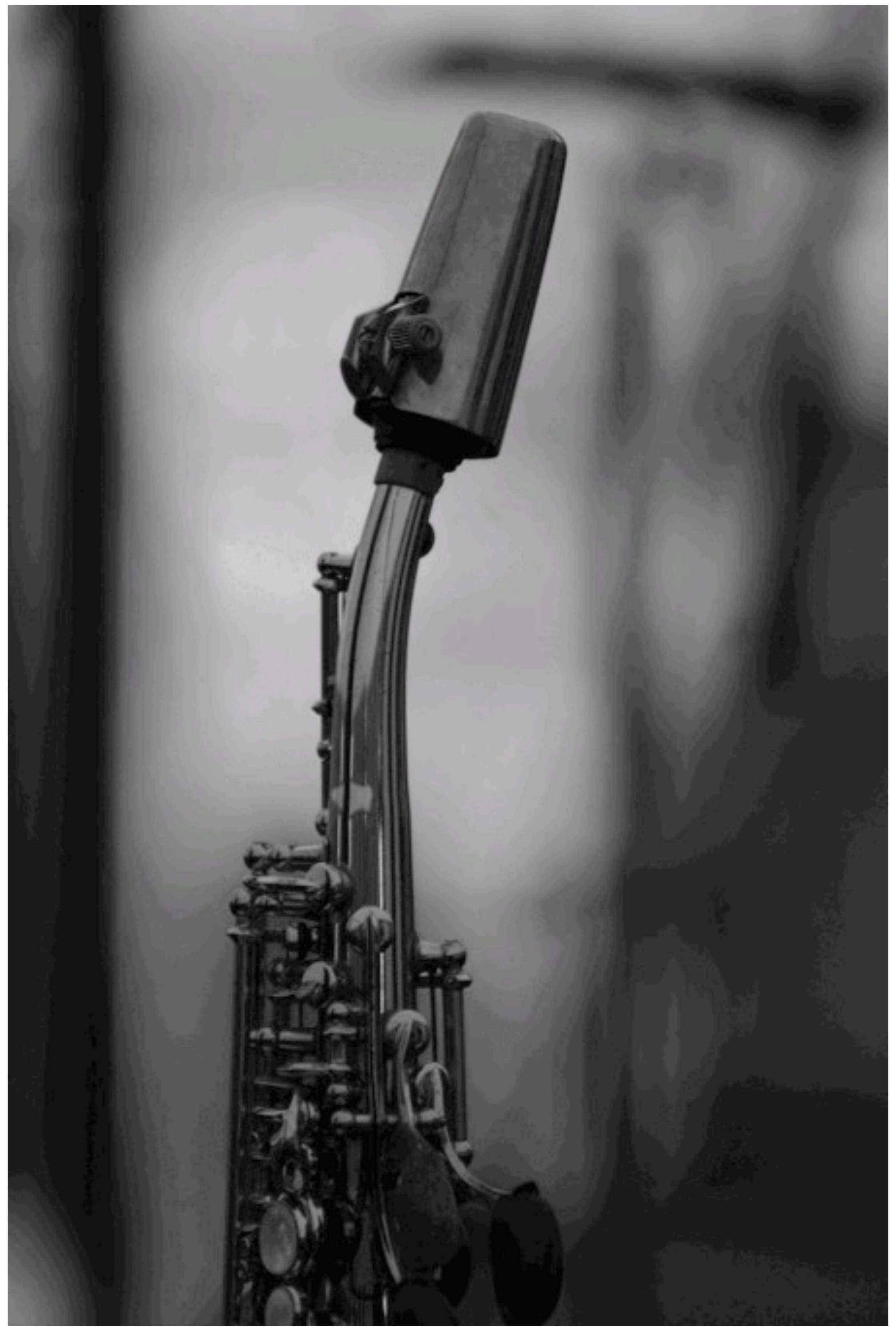
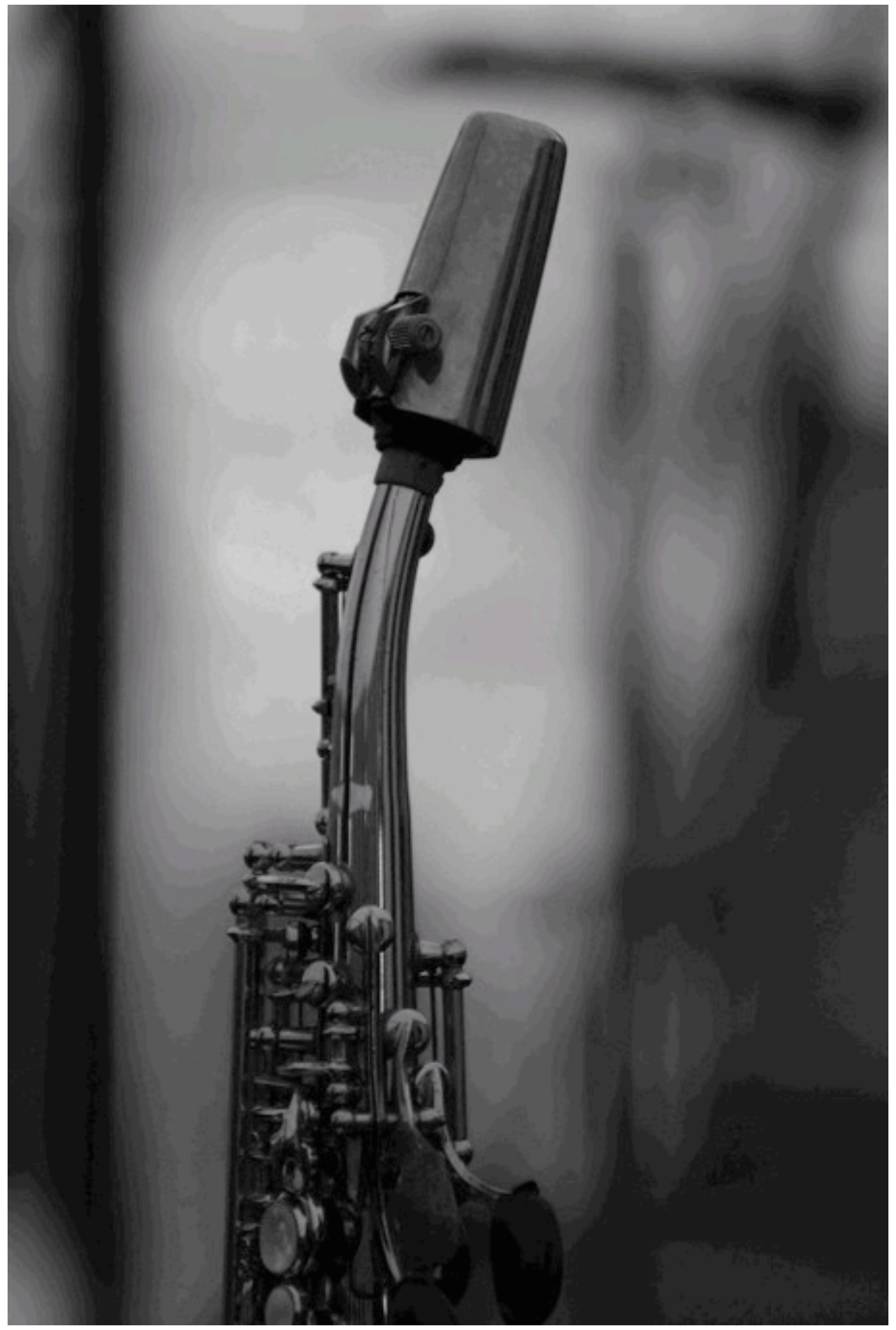
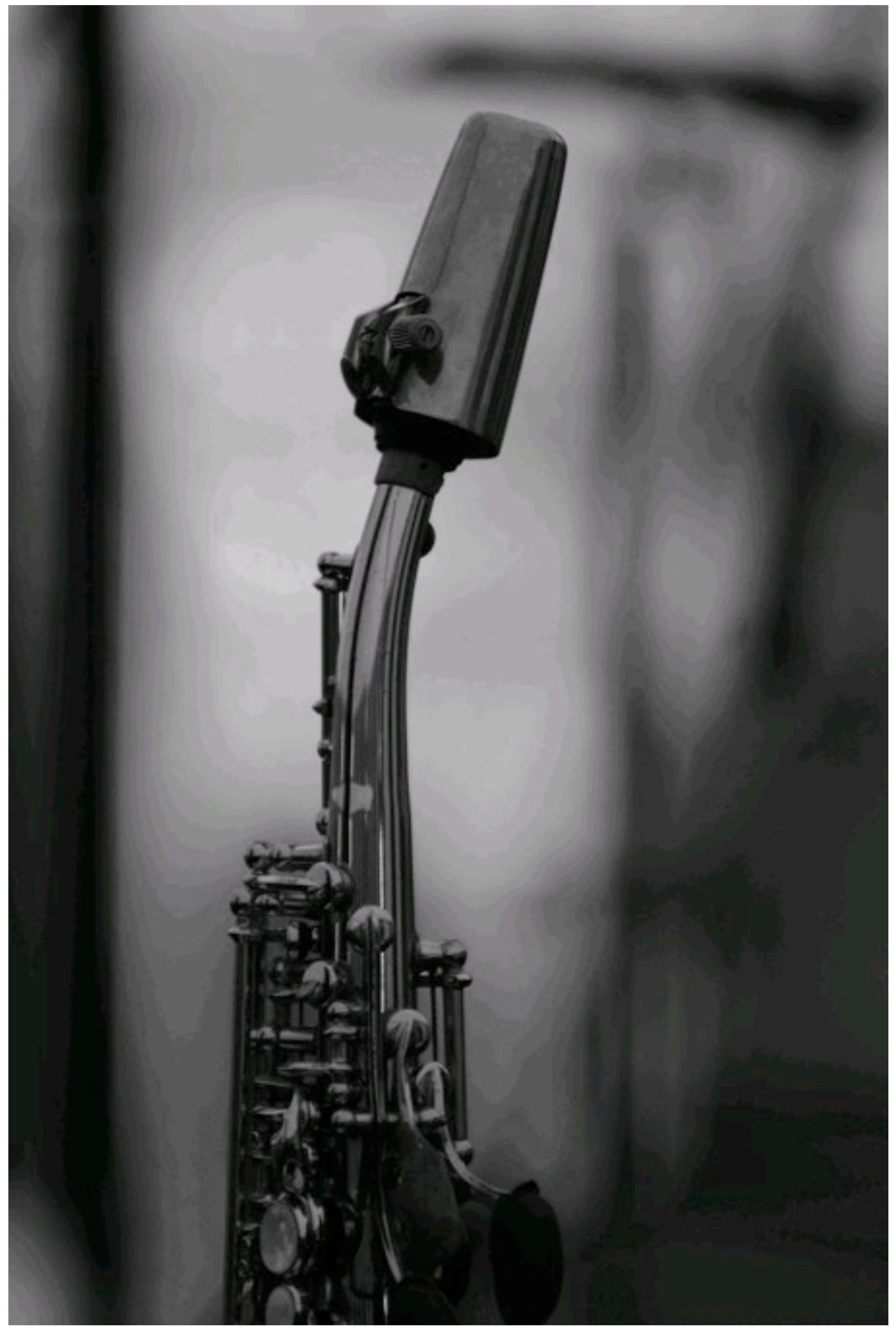
Basic Per Pixel Op

```
Load Image
for y in 0 to Image Height :
 for x in 0 to Image Width
   GetPixel at x,y
   ..... [do something]

Write out Image
```

Greyscale Algorithms (Cook 2009)

- Lightness method
  - max(r,g,b)+min(r,g,b)/2.0
- Average method
  - (r+g+b)/3
- Luminosity
  - 0.21*r+0.71*g+0.07*b

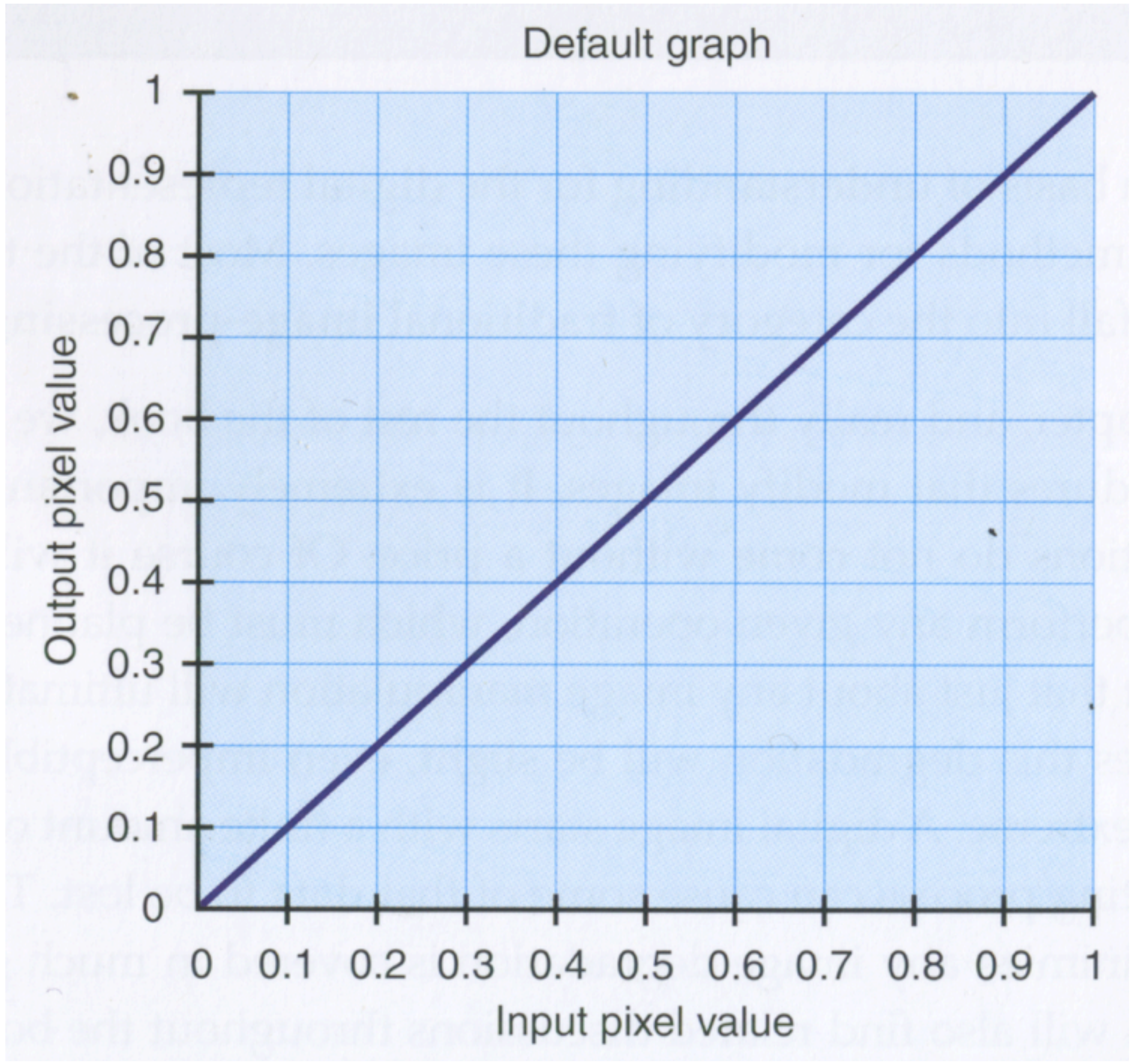
```
Load Image
for y in 0 to Image Height :
 for x in 0 to Image Width
   RGB=GetPixel at x,y
   grey=(R+G+B)/3.0
   Write grey to Pixel at x,y
```

Original	Lightness	Average	Luminosity
			

Look Up Table (LUT)

- An alternative approach to processing image data is to use a lookup table or LUT
- This is a simple graph that maps the input to output values
- This ranges from 0-1 on the input and 0-1 on the output
- The default graph would look like this

Default LUT



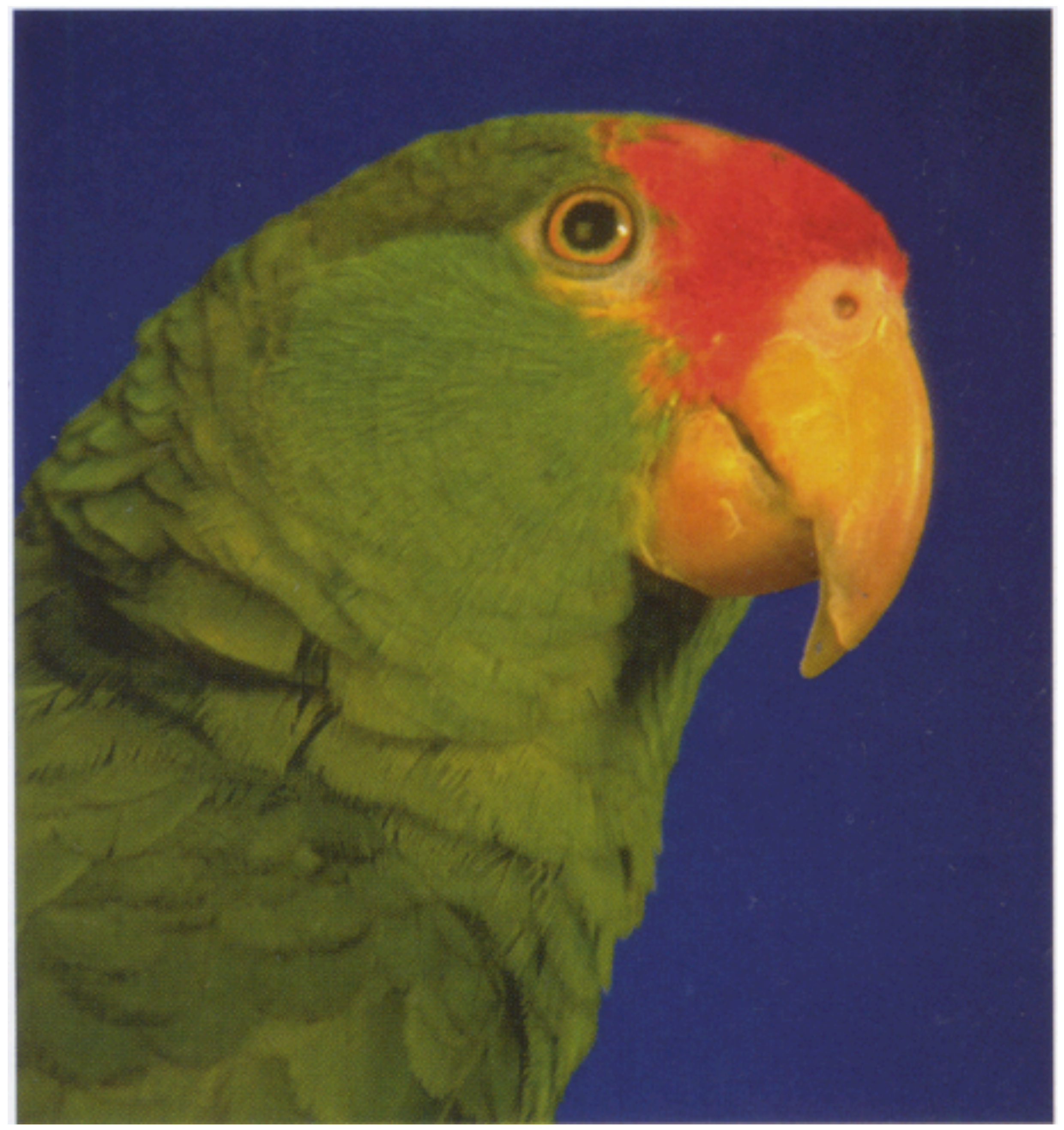
Images from Brinkmann 2003

Brightness (or Exposure or Gain)

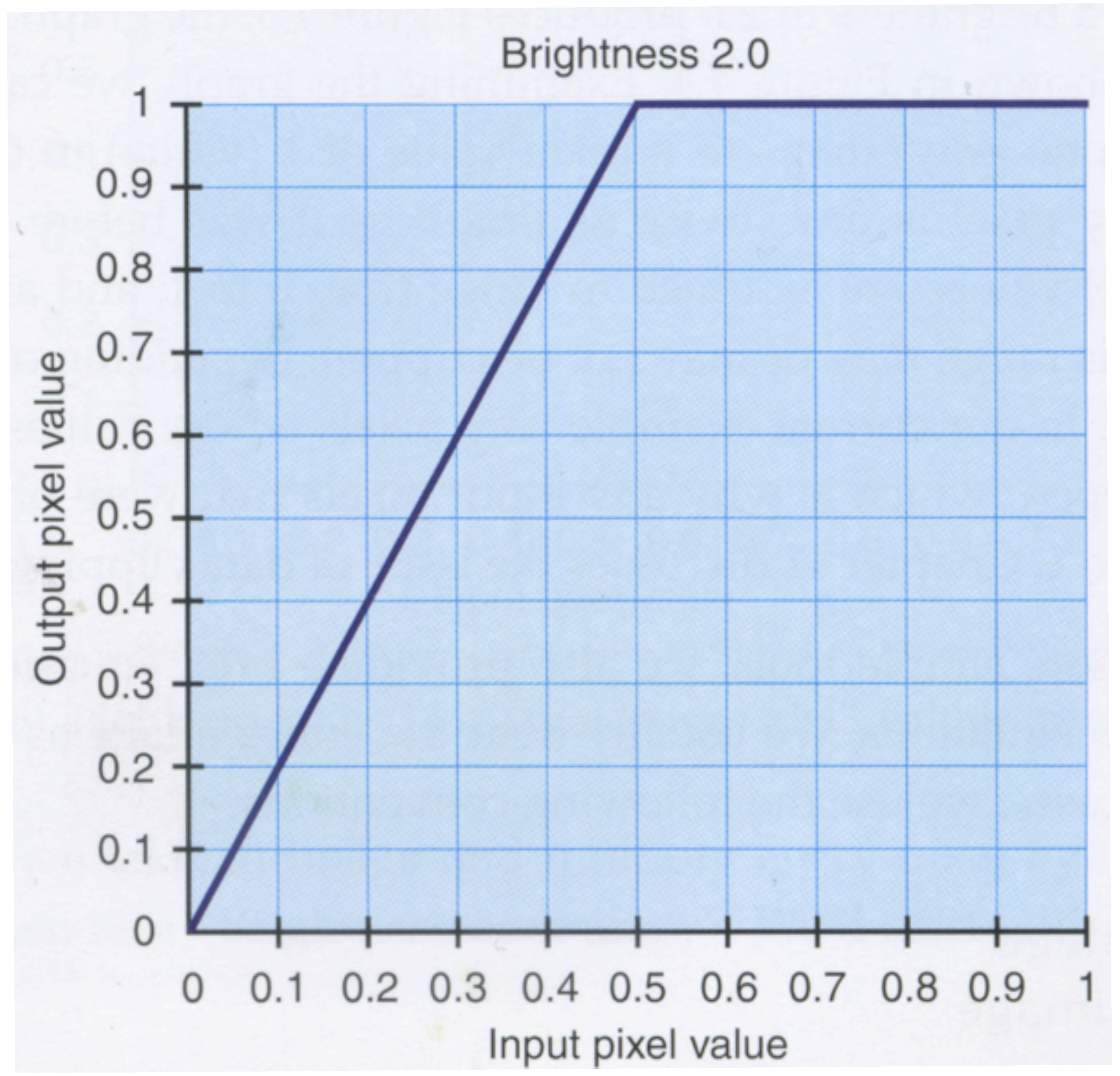
- If we take each pixel and multiply it by a single value (as shown in the earlier slides)
- We could implement a brightness option
- For example a brightness of 2.0 can be expressed as follows

I = input Image
O = output Image
O= I x 2.0

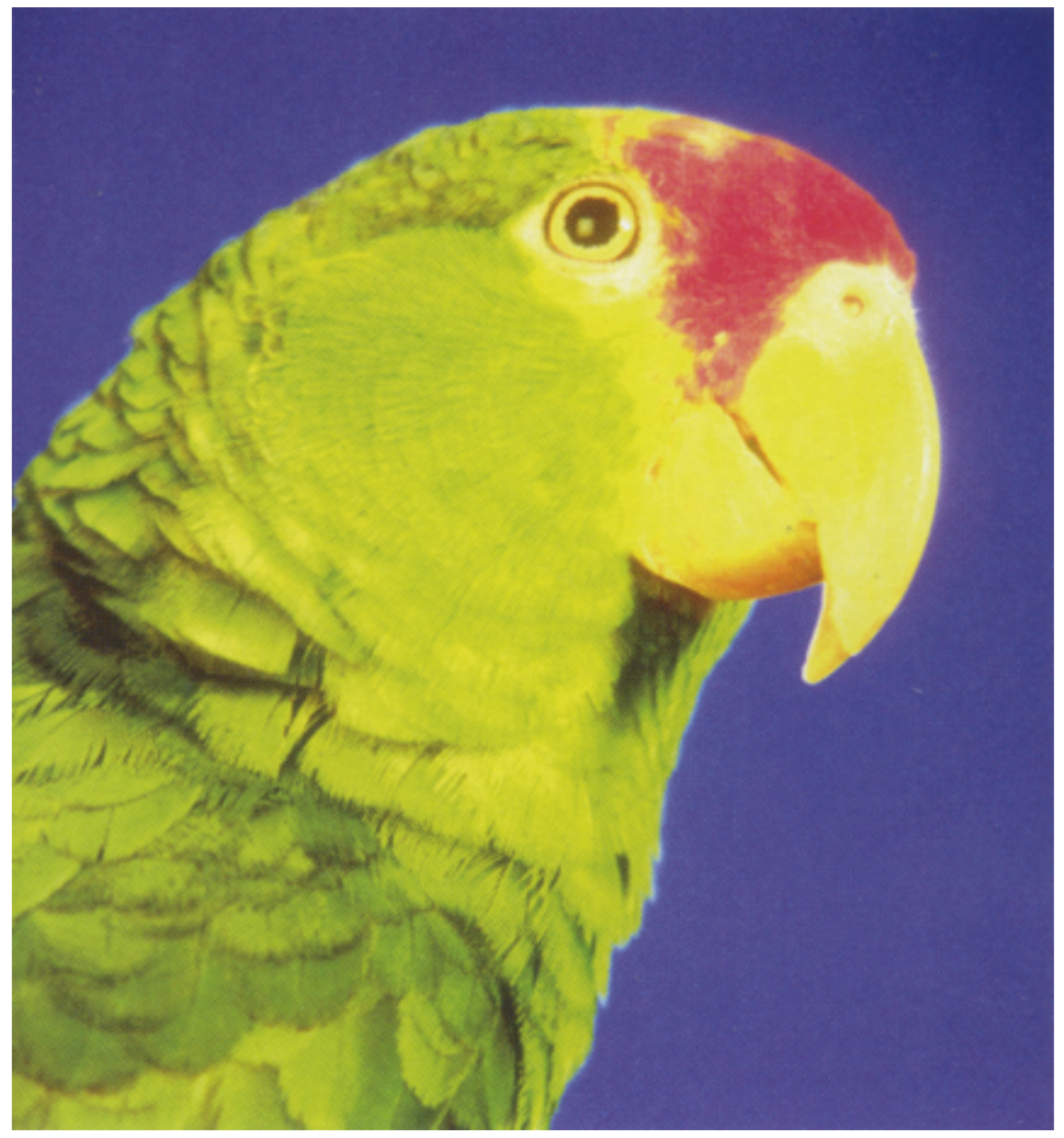
Brightness (or Exposure or Gain)



Input



x2.0



output

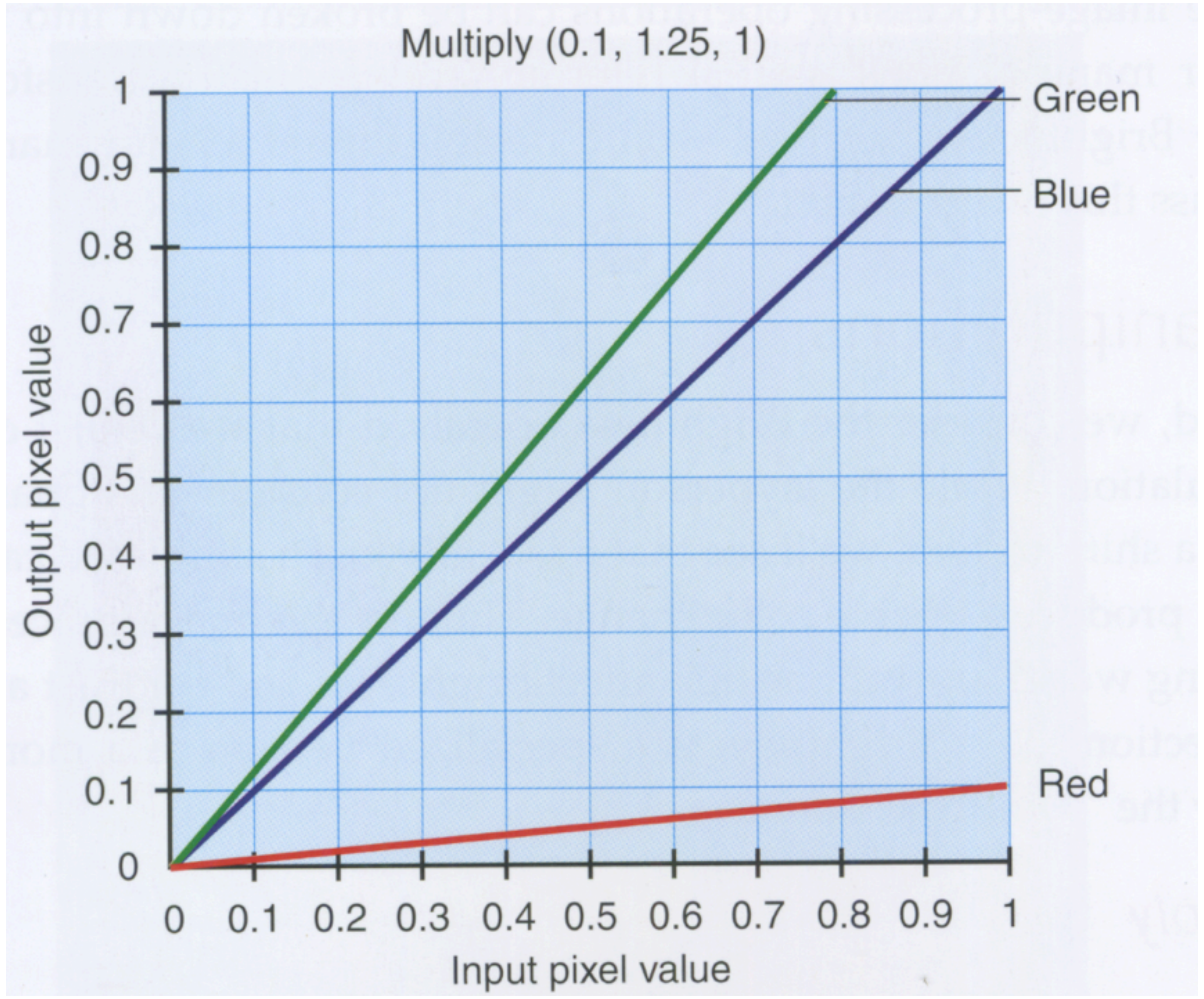
Images from Brinkmann 2003

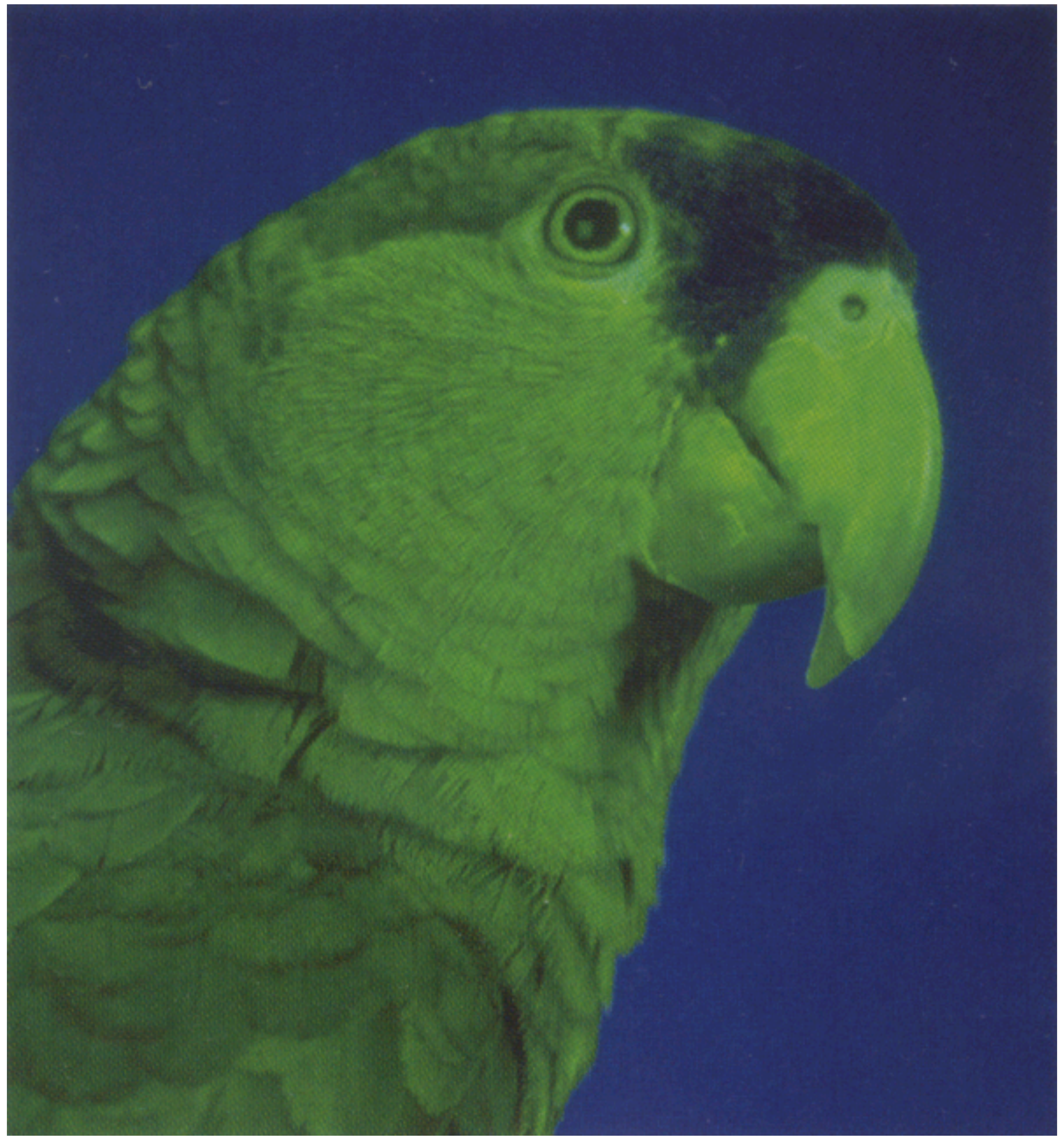
RGB Multiply

- In the previous example we multiply each of the tuple values by 2.0
- Usually we would create a LUT for each of the individual channels and apply different values to each channel
- In the following example we will apply

$$O_r = I_r * 0.1$$
$$O_g = I_g * 1.25$$
$$O_b = I_b$$

RGB Multiply





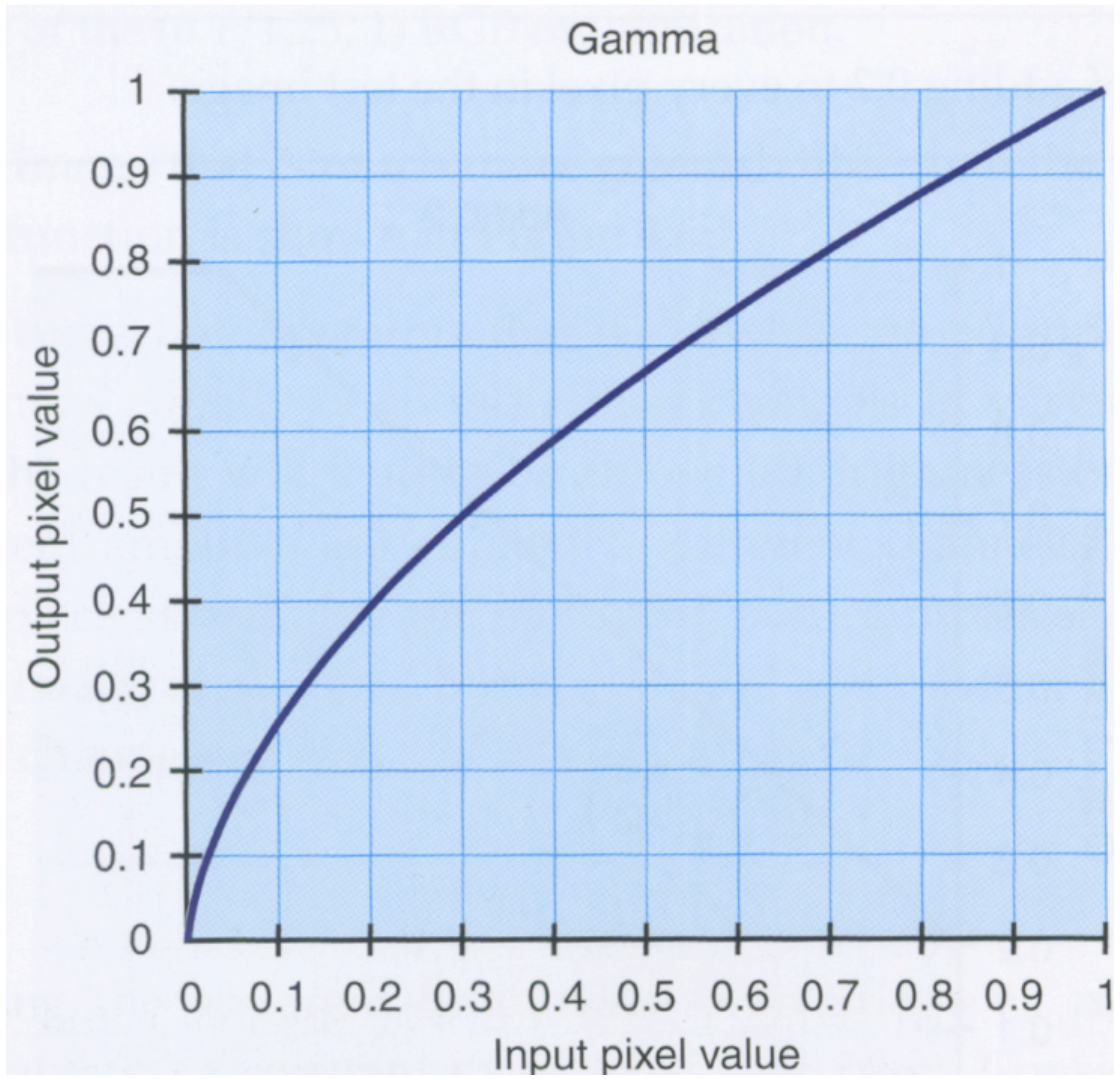
Images from Brinkmann 2003

Non-Linear LUT

- In the previous examples all of the graphs have been linear (straight lines)
- In many cases we actually want to use a curve or non-linear LUT
- A simple example of this is gamma correction
- This can be expressed using the equation

$$O = I^{1/\mathrm{Gamma}}$$

Gamma Correction

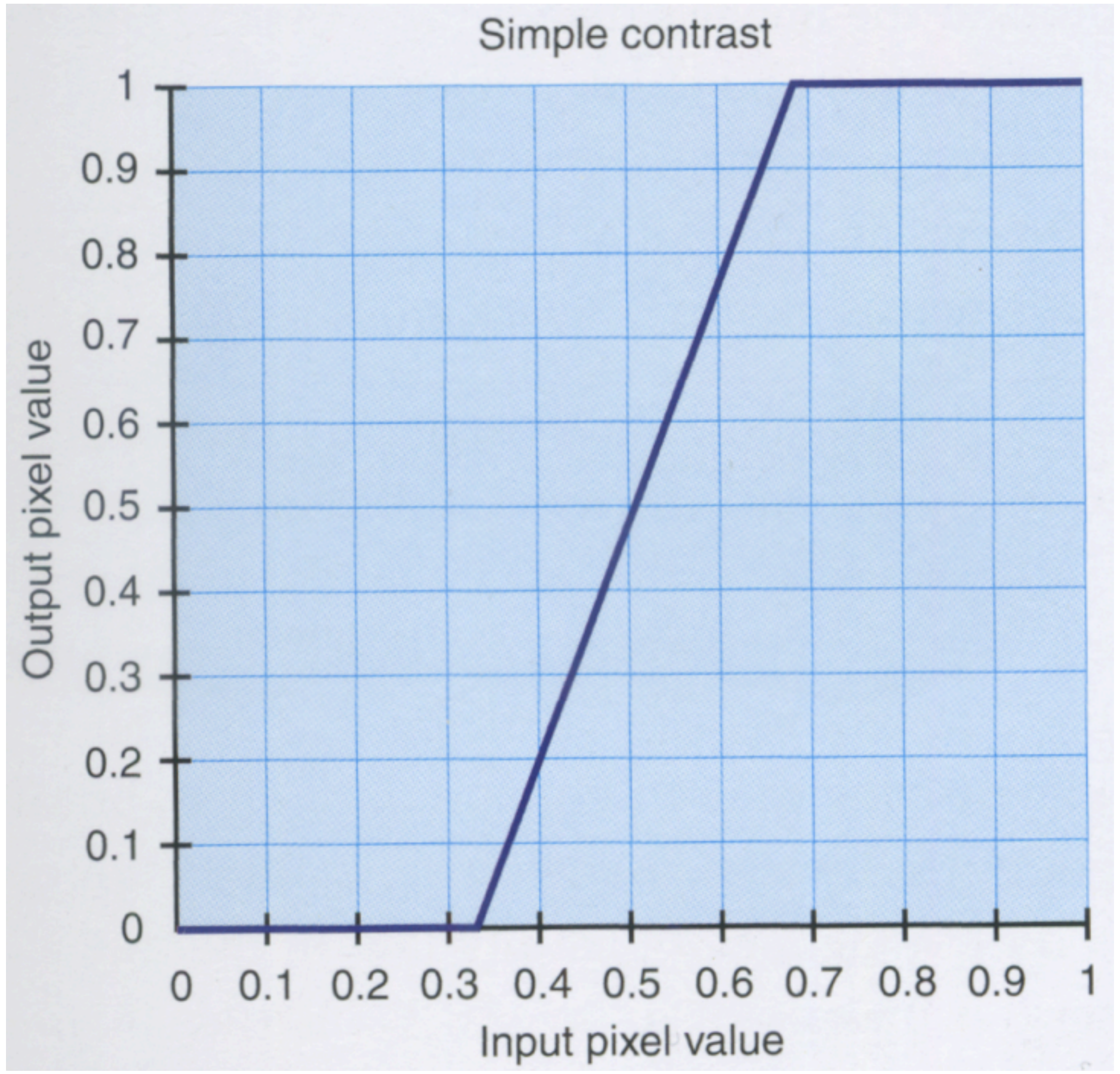


Images from Brinkmann 2003

Gamma Correction

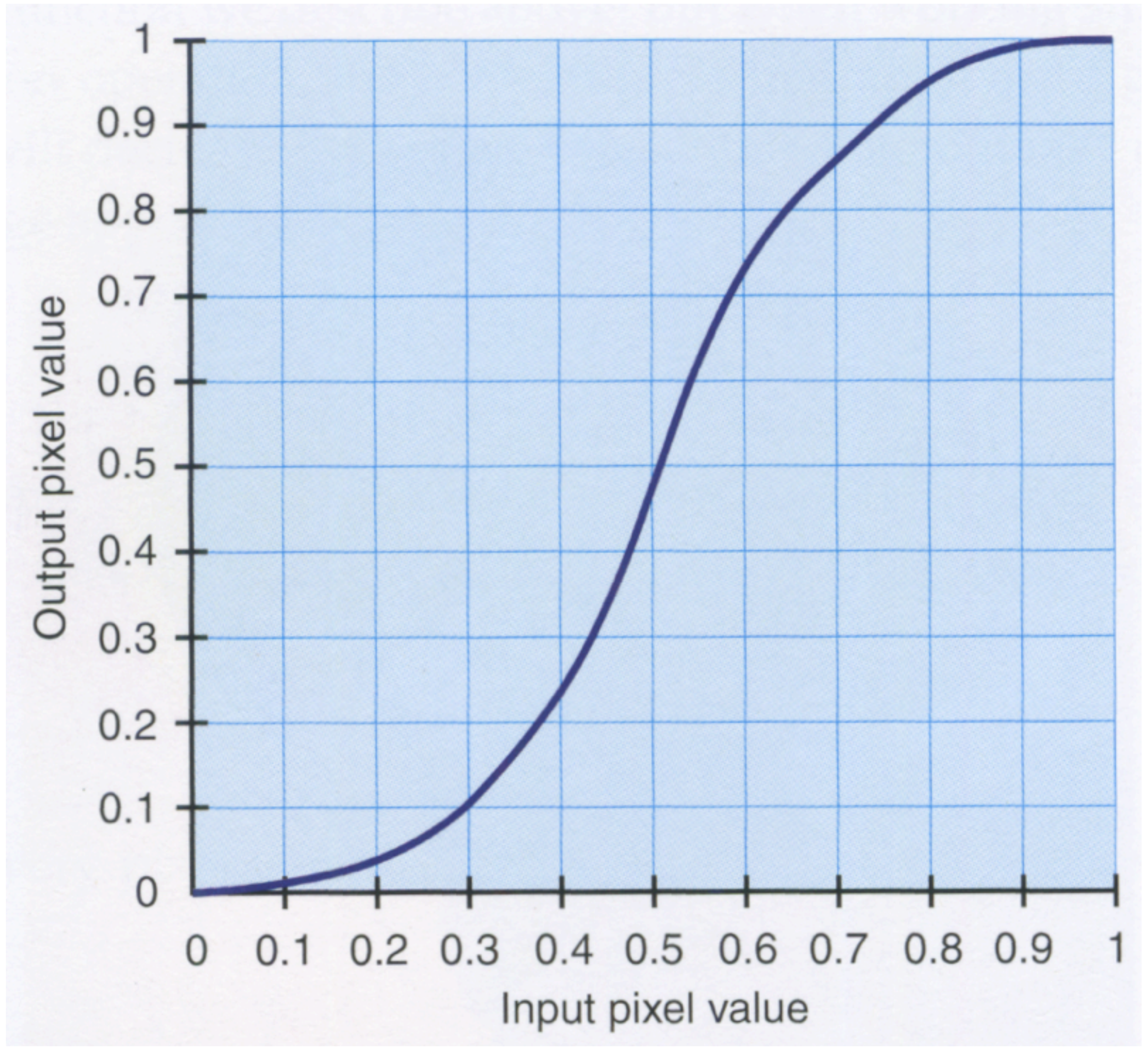
- Gamma correction works well because if you examine the graph you will see
  - When you raise 0 to any power it remains 0 ($0^3 = 0$)
  - When you raise 1 to any power it remains 1 ($1^3 = 0$)

Simple Contrast



Images from Brinkmann 2003

Smooth Contrast



Images from Brinkmann 2003

Alpha Compositing

- alpha compositing is the process of combining an image with a background to create the appearance of partial transparency.
- In order to correctly combine these image elements, it is necessary to keep an associated matte for each element.
- This matte contains the coverage information - the shape of the geometry being drawn
- This allows us to distinguish between parts of the image where the geometry was actually drawn and other parts of the image which are empty.

The Alpha Channel

- To store this matte information, the concept of an alpha channel was introduced by A. R. Smith (1970s)
- Porter and Duff then expanded this to give us the basic algebra of Compositing in the paper "Compositing Digital Images" in 1984.

The Alpha Channel

"A separate component is needed to retain the matte information, the extent of coverage of an element at a pixel.
In a full colour rendering of an element, the RGB components retain only the colour. In order to place the element over an arbitrary background, a mixing factor is required at every pixel to control the linear interpolation of foreground and background colours.
In general, there is no way to encode this component as part of the colour information.
For anti-aliasing purposes, this mixing factor needs to be of comparable resolution to the colour channels.
Let us call this an alpha channel, and let us treat an alpha of 0 to indicate no coverage, 1 to mean full coverage, with fractions corresponding to partial coverage."

Porter & Duff 84

Alpha Channel

- In a 2D image element which stores a colour for each pixel, an additional value is stored in the alpha channel containing a value ranging from 0 to 1.
- 0 means that the pixel does not have any coverage information; i.e. there was no colour contribution from any geometry because the geometry did not overlap this pixel.
- 1 means that the pixel is fully opaque because the geometry completely overlapped the pixel.
- It is important to distinguish between the following

black = (0,0,0,1)

clear = (0,0,0,0)

Pre multiplied alpha

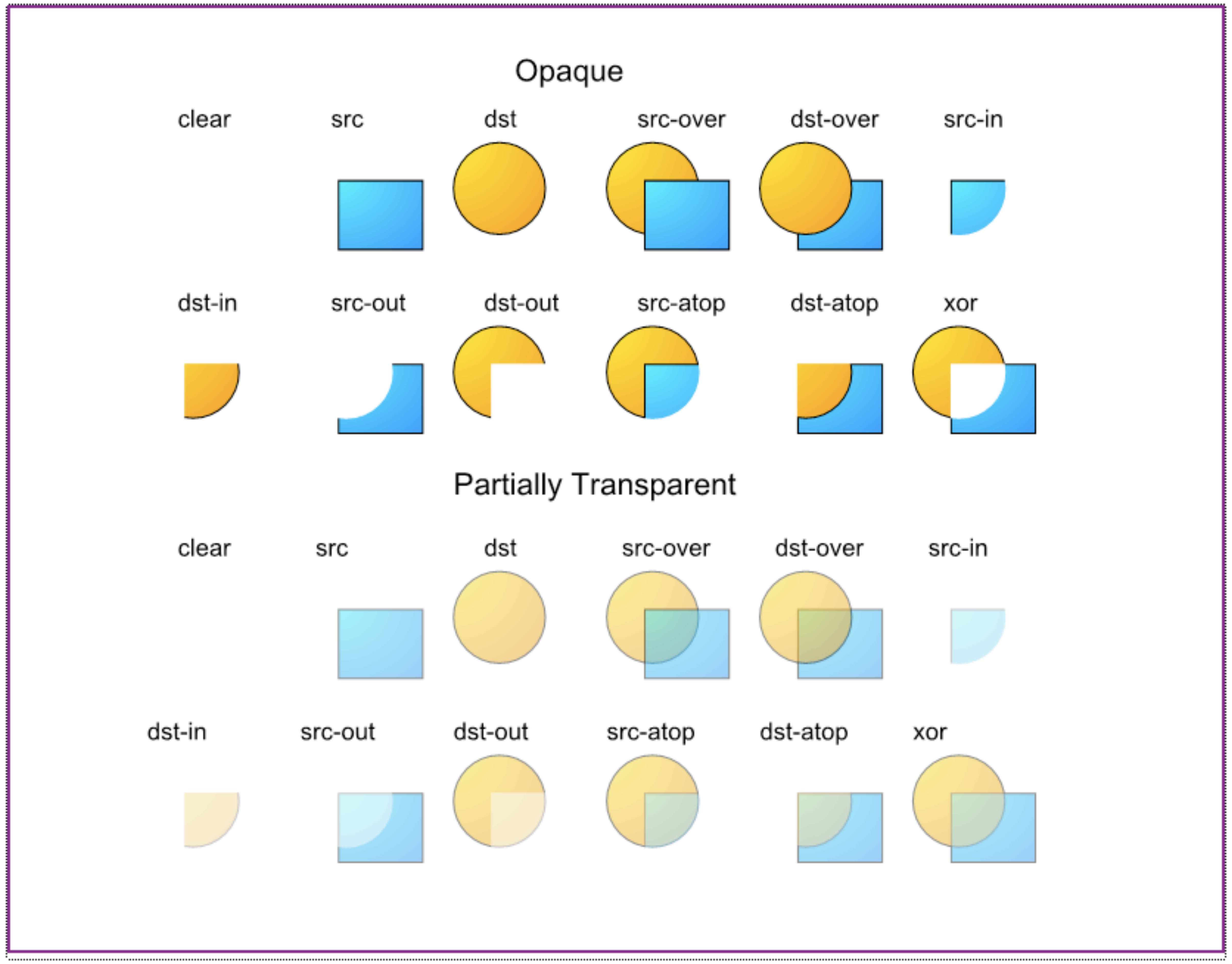
*"What is the meaning of the quadruple (r,g,b,a) at a pixel?
How do we express that a pixel is half covered by a full red object?
One obvious suggestion is to assign (1,0,0,.5) to that pixel: the .5 indicates the coverage and the (1,0,0) is the colour.
There are a few reasons to dismiss this proposal, the most severe being that all compositing operations will involve multiplying the 1 in the red channel by the .5 in the alpha channel to compute the red contribution of this object at this pixel.
The desire to avoid this multiplication points us to a better solution, storing the pre-multiplied value in the colour component, so that (.5,0,0,.5) will indicate a full red object half covering a pixel".*

Porter & Duff 84

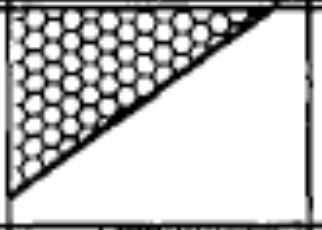
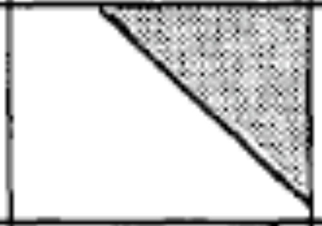
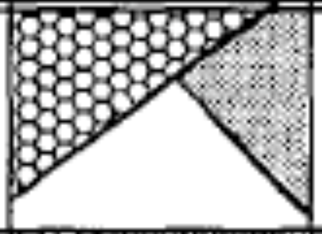
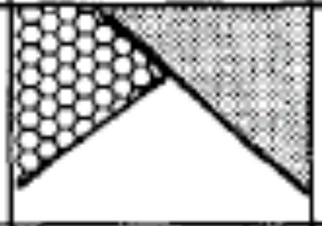
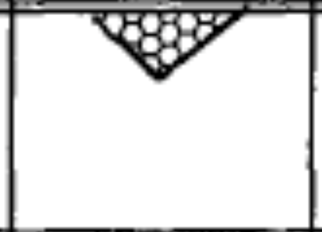
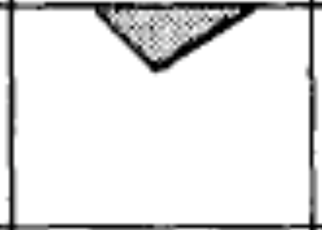
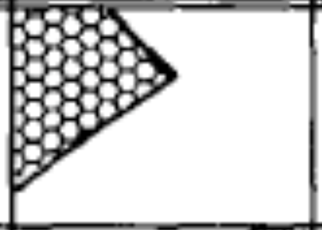
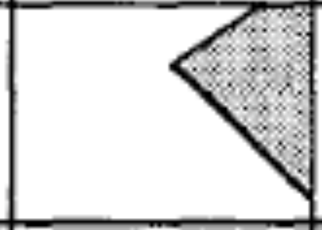
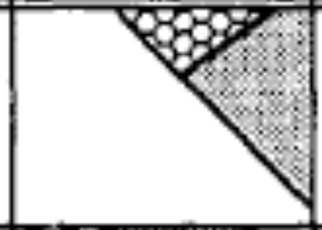
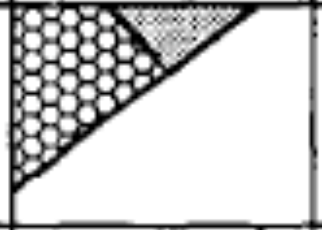
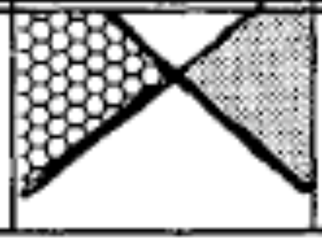
Pre multiplied alpha

- If an alpha channel is used in an image, it is common to also multiply the colour by the alpha value, in order to save on additional multiplication during the Compositing process.
- This is usually referred to as pre-multiplied alpha.
- Thus, assuming that the pixel colour is expressed using RGB triples,
- a pixel value of (0.0, 0.5, 0.0, 0.5) implies a pixel which is fully green and has 50% coverage.

Compositing Algebra



- Porter and Duff proposed a number of basic operations, which are performed on a per pixel, per RGB channel basis
- Most of these operations pre-suppose that the RGB channels have already been pre-multiplied by the alpha value

operation	quadruple	diagram	F_A	F_B
clear	(0,0,0,0)		**0**	**0**
A	(0,A,0,A)		**1**	**0**
B	(0,0,B,B)		**0**	**1**
A **over** B	(0,A,B,A)		**1**	$1-\alpha_A$
B **over** A	(0,A,B,B)		$1-\alpha_B$	**1**
A **in** B	(0,0,0,A)		α_B	**0**
B **in** A	(0,0,0,B)		**0**	α_A
A **out** B	(0,A,0,0)		$1-\alpha_B$	**0**
B **out** A	(0,0,B,0)		**0**	$1-\alpha_A$
A **atop** B	(0,0,B,A)		α_B	$1-\alpha_A$
B **atop** A	(0,A,0,B)		$1-\alpha_B$	α_A
A **xor** B	(0,A,B,0)		$1-\alpha_B$	$1-\alpha_A$

A over B

- The basic over operator is similar to the painters algorithm where the A element is placed over the B element.
- For each Pixel we do the following

$$C_o = C_a + C_b \times (1 - \alpha_a)$$

$$\alpha_o = \alpha_a + \alpha_b \times (1 - \alpha_a)$$

Other Functions

Function	Use	Maths
Atop	Add effects to foreground	$C_o = C_a \times \alpha_b + (C_b \times (1 - \alpha_a))$
ADD	Add mattes together	$C_o = C_a + C_b$
DIV		$C_o = C_a \div C_b$
Mult	Mask Elements	$C_o = C_a \times C_b$
Inside	Mask Elements	$C_o = C_a \times \alpha_b$
Sub	Subtract	$C_o = C_a - C_b$
outside	Mask Elements	$C_o = C_a \times (1 - \alpha_b)$
xor		$C_o = C_a \times (1 - \alpha_b) + C_b \times (1 - \alpha_a)$

Unary operators

- To assist in dissolving and colour balancing Porter and Duff also suggested the following operations

$$\text{darken}(A, \phi) \equiv (\phi r_A, \phi g_A, \phi b_A, \alpha_A)$$
$$\text{dissolve}(A, \delta) \equiv (\delta r_A, \delta g_A, \delta b_A, \delta \alpha_A)$$

Normally $0 \leq \phi, \delta \leq 1$ although none of the theory requires it

As ϕ varies from 1 to 0, the element will change from normal to complete blackness.

If $\phi > 1$ the element will be brightened.

As δ goes from 1 to 0 the element will gradually fade from view

Luminescent objects

- Luminescent objects, which add colour information without obscuring the background, can be handled with the introduction of a opaqueness factor $\omega, 0 \leq \omega \leq 1$

$$\text{opaque}(A, \omega) \equiv (r_A, g_A, b_A, \omega\alpha_A)$$

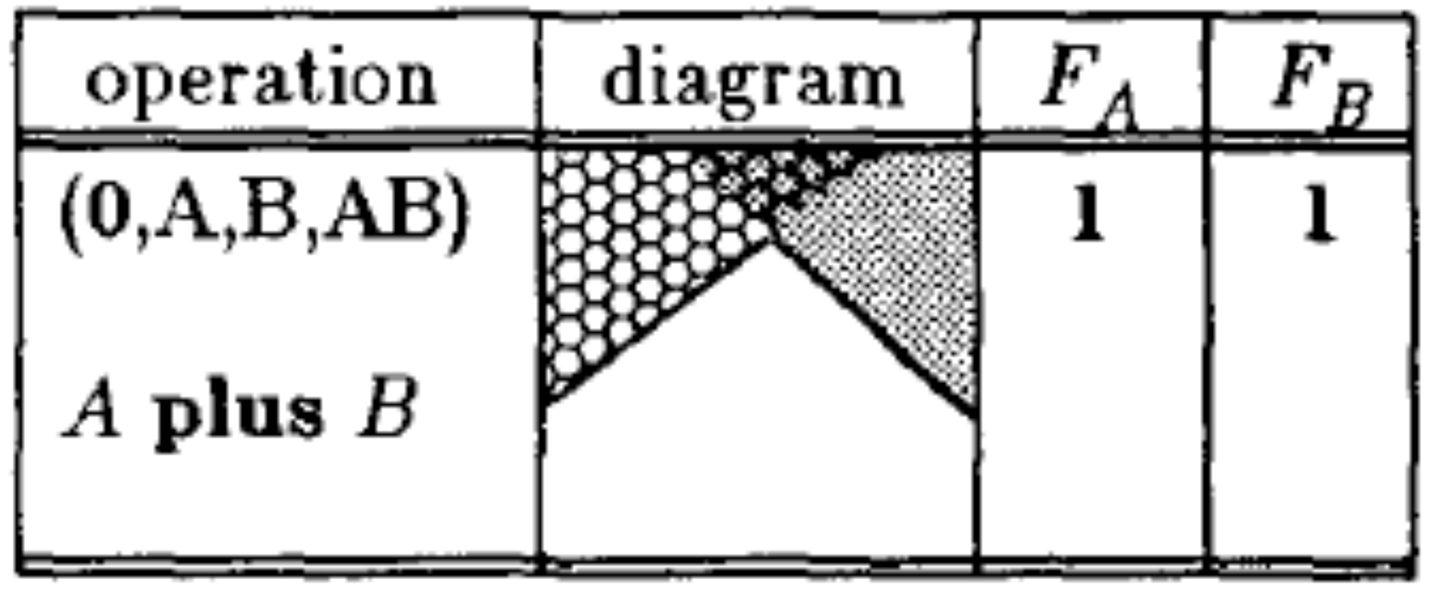
As ω varies from 1 to 0, the element will change from normal coverage over the background to no obscuratior

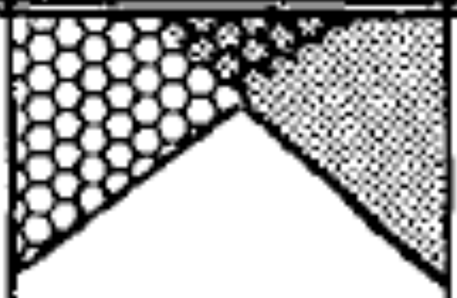
The Plus Operator

- The expression A plus B holds no notion of precedence in any area covered by both pictures; the components are simply added.
- This allows us to dissolve from one picture to another by specifying

$$\text{dissolve}(A, \alpha) \text{ plus } \text{dissolve}(B, 1 - \alpha)$$

- In terms of the binary operators above, plus allows both pictures to survive in the subpixel area AB.



operation	diagram	F_A	F_B
(0,A,B,AB) *A* **plus** *B*		1	1

References

- John. D. Cook (2009). Three algorithms for converting color to grayscale. Aug 24 2009. The Endeavour [online]. [Accessed 27 September 2010]. sAvailable from: <http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/>.
- Thomas Porter and Tom Duff, Compositing Digital Images, Computer Graphics, 18(3), July 1984,
- http://en.wikipedia.org/wiki/Alpha_transparency
- Brinkmann R, “The art and Science of Digital Compositing” 2nd Edition 2008 Morgan Kaufmann.