lecture 20

Image Compositing

- chroma keying
- alpha
- F over B
- OpenGL blending
- chroma keying revisited: "pulling a matte"

Organization of Course

1: Viewing transformations

2: Visibility, geometry modelling

3: Rendering: light, material, texture, transparency

Transparency is a mix of rendering and image capture/display. It is a bridge between parts 3 and 4 of the course.

4: Image Capture and Display

Many computer graphics techniques use real images in some way.

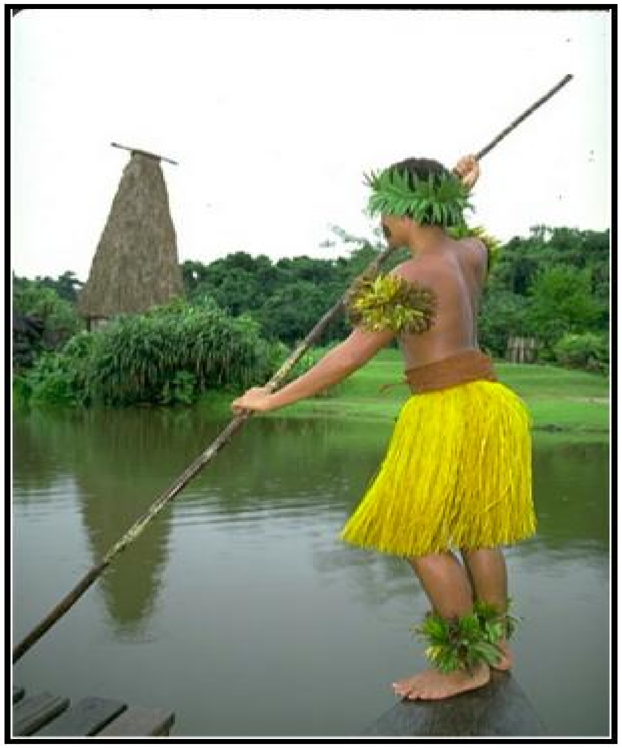
We have seen several examples
- scanned 3D models
- texture mapping using photos
- environment mapping

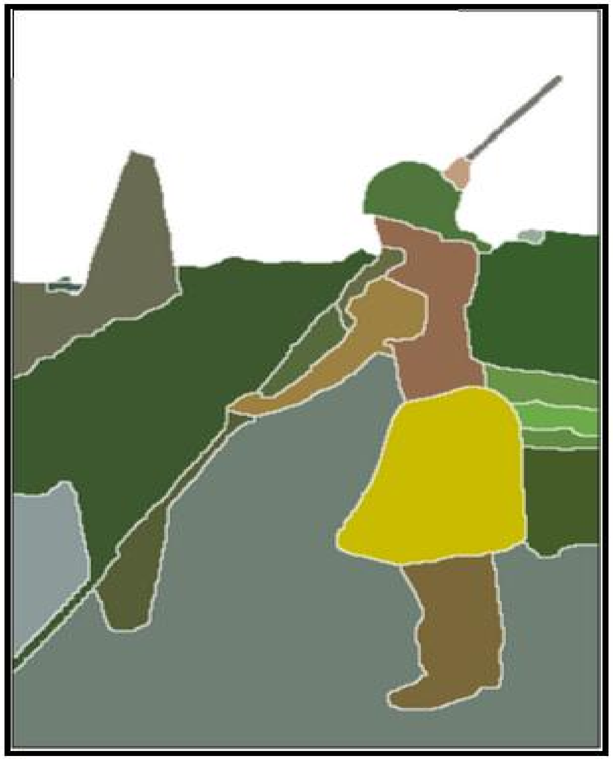
Let's start today's lecture with another example.

Image Segmentation

Classic computer (and human) vision problem:

Partition an image into regions.   It is a difficult problem (and not so well defined).

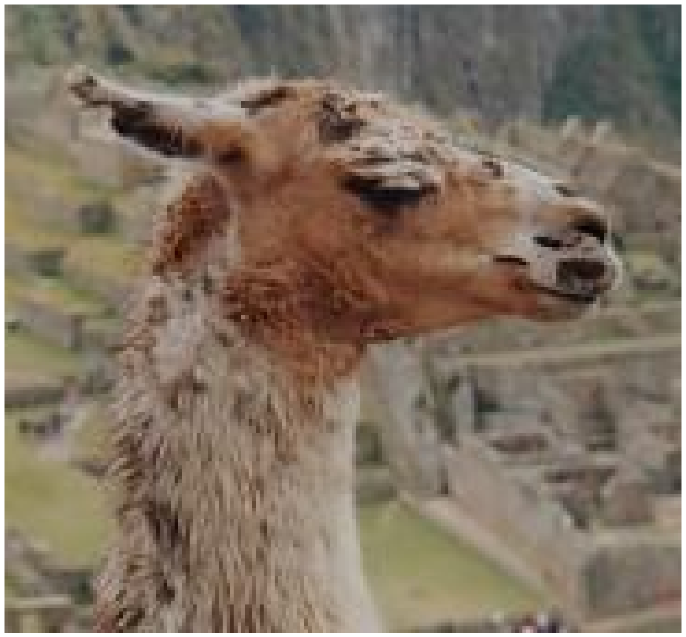


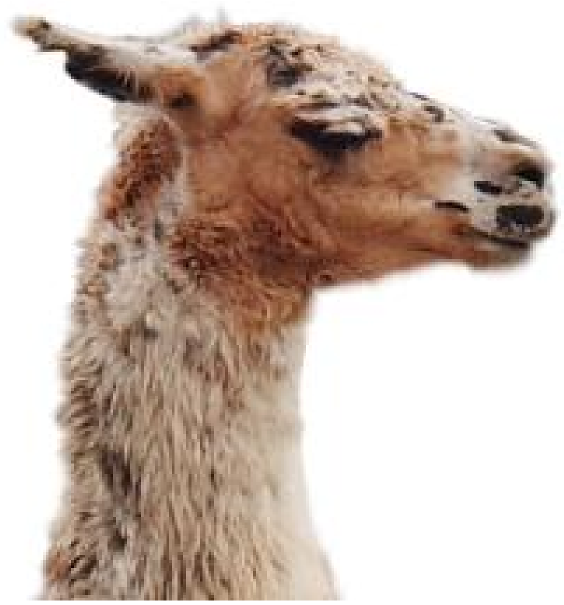


http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html

Specific version of segmentation:

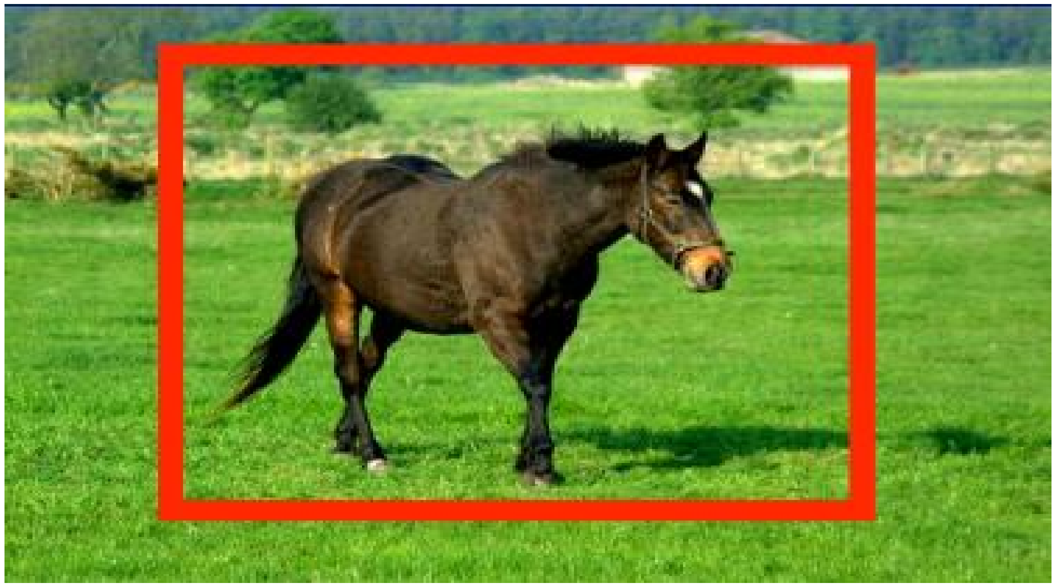
Given an image, partition it into a foreground and a background.

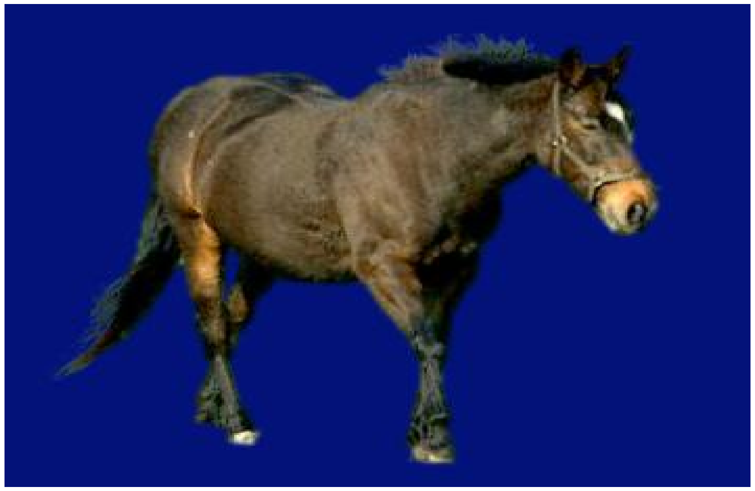

input

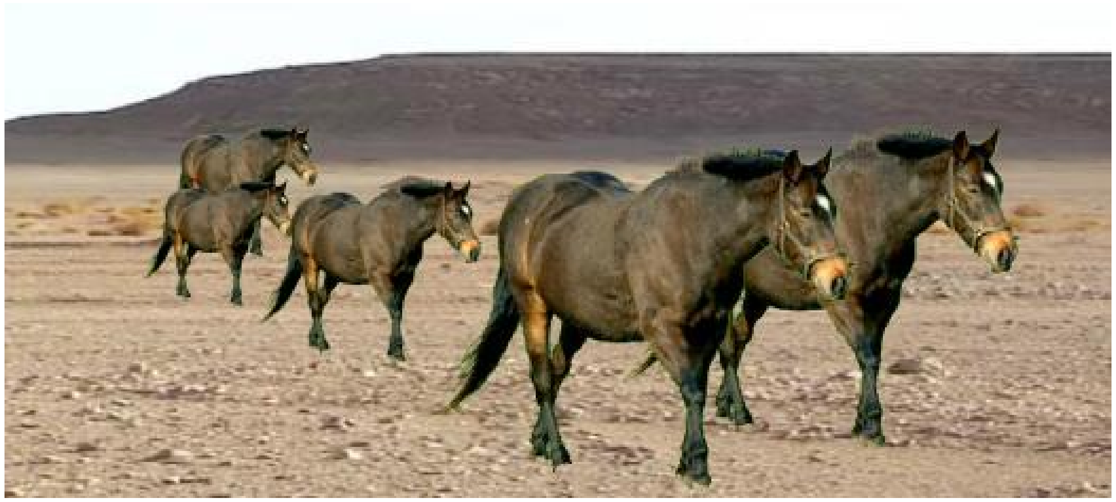

foreground

http://www.cc.gatech.edu/~dellaert/07F-Vision/Schedule_files/10-LazySnapping.ppt.pdf

Computer graphics application: the foreground can then be pasted over a different background ("compositing")


input


(semi) automatic segmentation

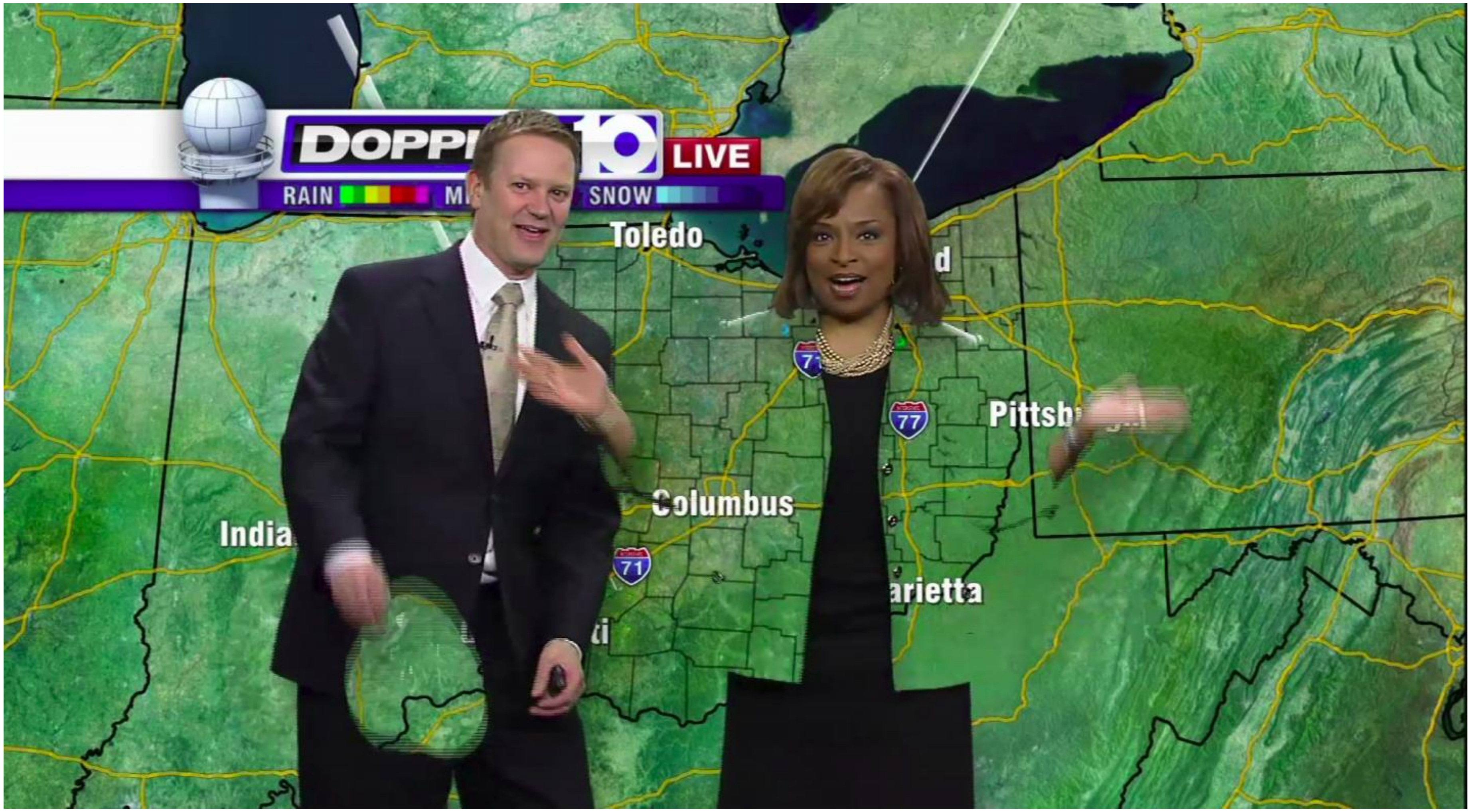

output (composite with new background)

This is an old idea e.g. chroma-keying

(green or blue screen)

It doesn't always work.   (see video link)

http://www.10tv.com/content/stories/2014/03/17/tracy-townsend-wears-green-disappears.html



General Approach

Step 1: Take picture of background B (not necessarily green screen)

Step 2: Take image/video of foreground character in front of background (F over B)

Step 3: // Compute foreground mask

```
For each pixel,
     if  (F over B)(x,y) ==  B(x,y)
          mask(x,y) = 0   // background
     else mask(x,y) = 1   //  foreground
```

Step 4: // Write foreground image over a new background Bnew

```
For each pixel (x,y)
     if  mask(x,y) == 1
        I(x,y)  =  F(x,y)
     else   I(x,y)  =  Bnew(x,y)
```

Why doesn't it always work?

- Cast shadows (foreground object can change background)

- Interreflections (green screen can reflect, so foreground takes on color of background)

- Foreground object might happen to have same color as background (in Step 3) -- see green screen example 2 slides ago

- Soft edges become hard (mask) e.g Hair and furry object boundaries are difficult to model with a binary mask.

Now let's look at a more general situation....

lecture 20

Image Compositing

- chroma keying
- alpha
- F over B
- OpenGL blending
- Chroma keying revisited: "pulling a matte"

Partially occupied pixels & "alpha"

Think of a pixel as a little square. The occupancy or coverage of a pixel is called "alpha".

$\alpha = 0$ means not occupied at all (transparent).

$\alpha = 1$ means fully occupied (opaque)

$0 < \alpha < 1$ means partially occupied

In representing RGB images is common to include a 4th component to indicate how much of the pixel is occupied, so we have RGBA. Typically one uses 8 bits for each "channel" so this gives 32 bits per pixel.

Examples of RGBA

(0, 0, 0, 1) - black and opaque

(1, 0, 0, 1) - red and opaque

etc.

(1, 1, 1, 1) - white and opaque

In the following, I used "premultiplied" notation (explained soon)

(.5, 0, 0, .5) - red and 50% transparent

(.5, .5, .5, .5) - white and 50% transparent

(.1, .1, .1, .5) - dark grey and 50% transparent

(.1, .1, .1, .1) - white and 10% opaque (90% transparent)

(0, 0, 0, 0) - color undefined, 100% transparent

I will sometimes write RGB and sometimes rgb.
The reasons will be explained later ("premultiplied values")

To give you a flavour of what's to come....

Q: How do we darken a pixel without changing its opacity ?

A: darken($I_{rgb\alpha}$, ϕ) $= (\phi I_r, \phi I_g, \phi I_b, I_\alpha)$

Q: How do we change the opacity α of a pixel without changing the underlying color (sometimes called "dissolve") ?

dissolve($I_{rgb\alpha}$, δ) $= (\delta I_r, \delta I_g, \delta I_b, \delta I_\alpha)$

Where do alpha values come from ?

In OpenGL, we can define surfaces as partially transparent.

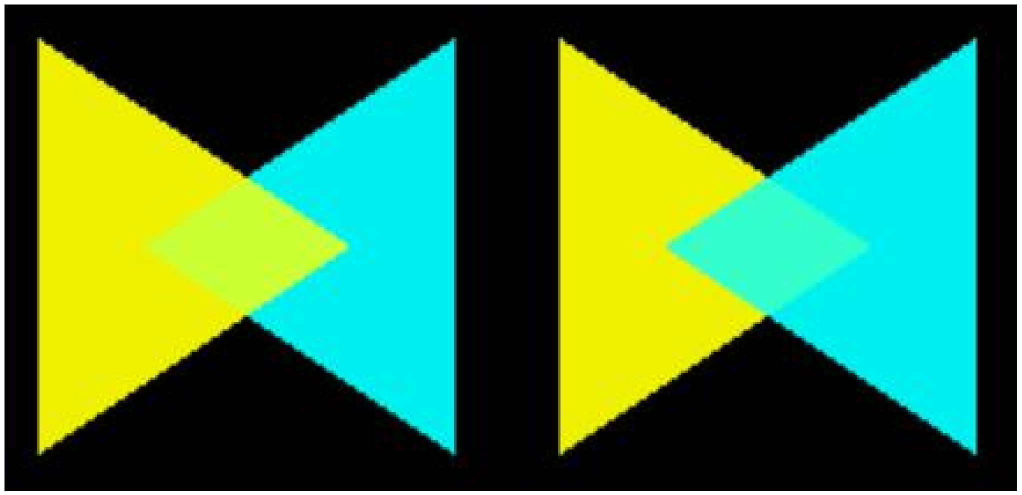
e.g.

```
  diffuse_material =  [ 1,  0,  0,   0.5 ]
  glMaterial(GL_FRONT,  GL_DIFFUSE,  diffuse_material)
  drawPolygon()
```

The material has a red color with **50% transparency**.

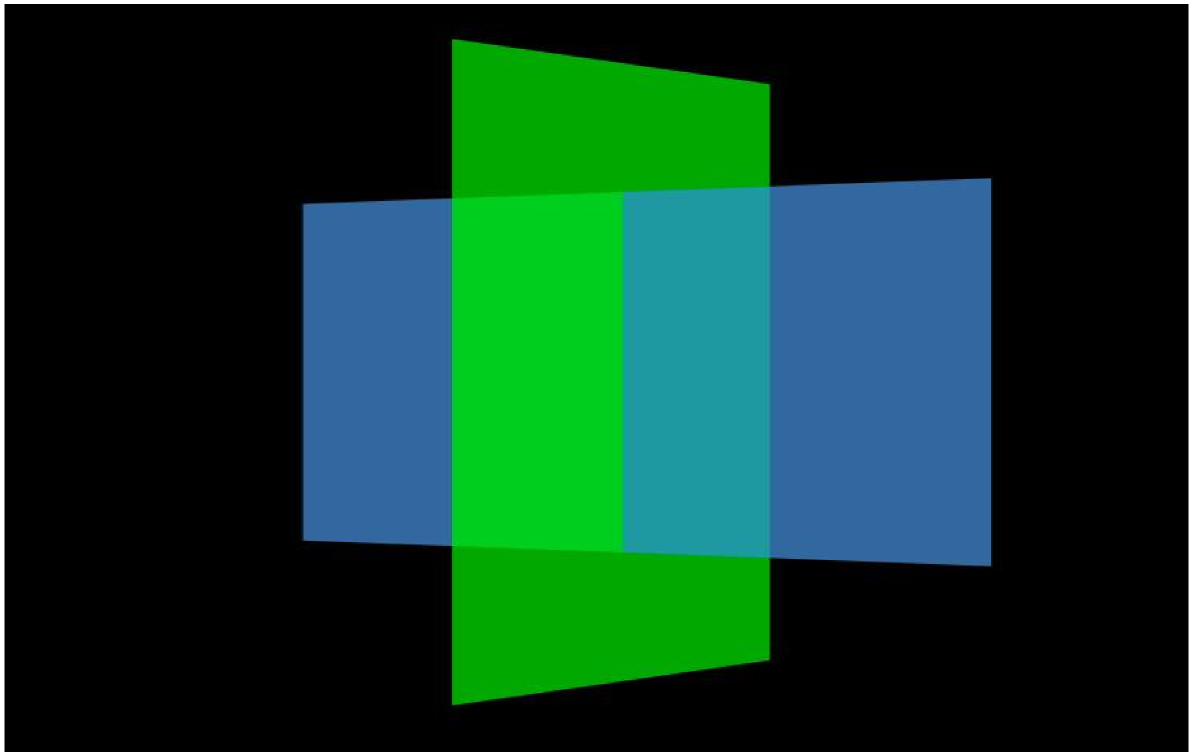
```
//  glEnable(GL_BLEND)
//  glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)  //   explain later
//  glDisable(GL_DEPTH_TEST)

def drawYellowTriangle():
   glBegin (GL_TRIANGLES)
   glColor4f(1.0, 1.0, 0.0, 0.75)       #  yellow
   glVertex3f(0.1, 0.9, 0.0)
   glVertex3f(0.1, 0.1, 0.0)
   glVertex3f(0.7, 0.5, 0.0)
   glEnd()

def drawCyanTriangle():                    #  cyan
   glBegin (GL_TRIANGLES)
   glColor4f(0.0, 1.0, 1.0, 0.75)
   glVertex3f(0.9, 0.9, 0.0)
   glVertex3f(0.3, 0.5, 0.0)
   glVertex3f(0.9, 0.1, 0.0)
   glEnd()

def drawMain():
   glPushMatrix()
   drawYellowTriangle()   //  right pair
   drawCyanTriangle()

   glTranslatef(-1,0, 0)
   drawCyanTriangle()
   drawYellowTriangle()    //  left pair
   glPopMatrix()
```

https://www.opengl.org/archives/resources/code/samples/redbook/alpha.c



In the previous example, all triangles were in the z=0 plane (and depth buffering was turned off). I just wanted to illustrate that the drawing order matters.

Here is another example which illustrates a more subtle point. For this example, there is no correct order to draw the two rectangles, since you cannot say that one rectangle is over another.



http://stackoverflow.com/questions/16774372/opengl-alpha-blending-and-object-independent-transparency

If you draw blue first, then green will be drawn over blue at each pixel. However, there are some pixels in which the green rectangle is behind the blue one. (Drawing the green first creates a similar problem.)

The solution is similar to the painter's algorithm: split one of the rectangles and draw them from far to near.

lecture 20

Image Compositing

- chroma keying
- alpha
- F over B
- OpenGL blending
- Chroma keying revisited: "pulling a matte"

F over B

Let's look at the "over" operation more formally.

How to put a foreground RGBA layer over a background RGBA layer?

I will use lower case "rgb" instead of RGB (for reasons to be explained later -- namely using "premultiplied" values).

Notation: Foreground $F_{rgb\alpha}$
Background $B_{rgb\alpha}$

Goal: How to compute a new RGBA layer which is the foreground layer over the background layer, i.e.

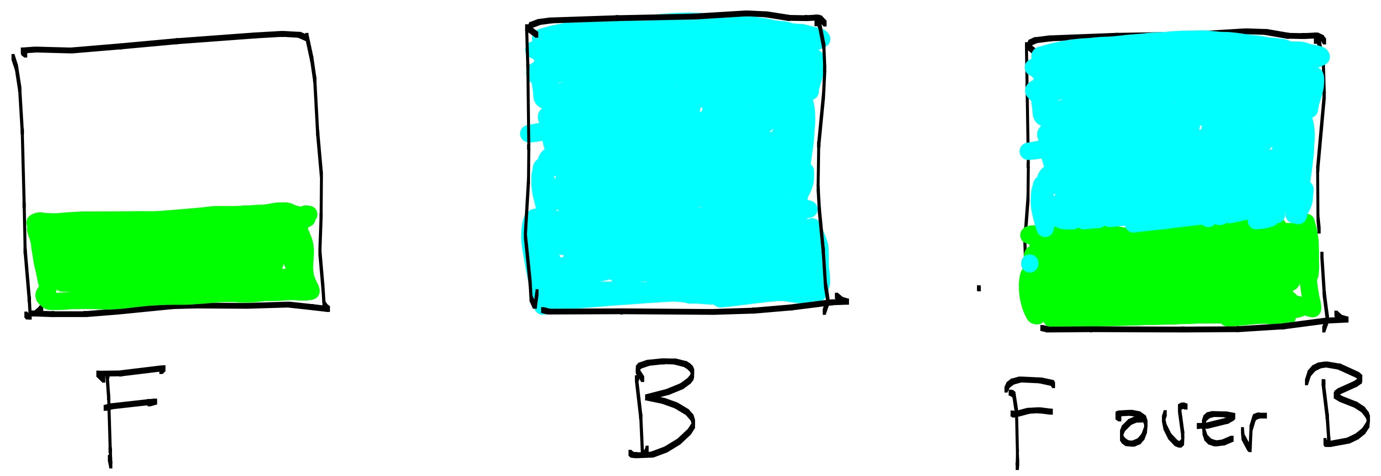
$$(F \text{ over } B)_{rgb\alpha} = ?$$

Special but common case (opaque background):

background is opaque, $B\alpha = 1$

foreground may be partly transparent, $0 < F\alpha < 1$

one pixel:



$$(F \text{ over } B)\alpha = F\alpha + (1 - F\alpha)\, B\alpha = 1$$

Let's not write out color yet.

I changed the slide order and content from the lecture.

More general case:

Background may be partly transparent, $0 <= B_\alpha <= 1$
Foreground may be partly transparent, $0 <= F_\alpha <= 1$

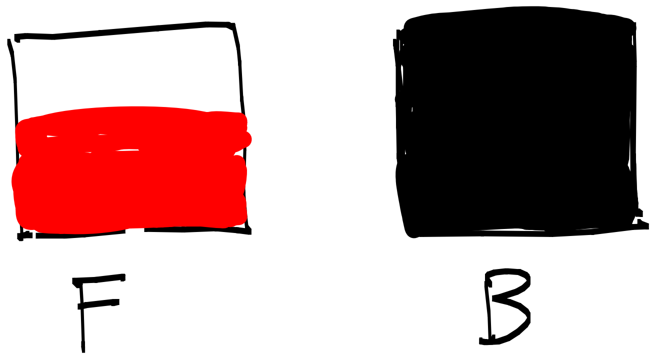
Again, given $F_{rgb\alpha}$, $B_{rgb\alpha}$, how do we define $(F \text{ over } B)_{rgb\alpha}$?

Note this is a per-pixel definition.

Example

Suppose the background color is black. Its RGB color is (0, 0, 0).

Suppose the foreground color is red.
We think of foreground RGB color as (1,0,0), e.g. glColor(1, 0, 0)
Suppose the foreground has $\alpha = 0.5$.



There are two ways to interpret a partially occupied pixel. First, the pixel is transparent. Second, the underlying surface may be opaque but it only covers part of the pixel because it is near the boundary of the surface. For the present discussion, we don't care which of these two situations is present. (The illustrations use the second.)

How should the RGBA values of the foreground pixel be interpreted/defined/represented ?

- You might argue it should be represented as

  (1, 0, 0, 0.5)

  since we have a red surface and the alpha value is 0.5.

- Or, you might argue that it should be represented as

  (0.5, 0, 0, 0.5)

  since the RGB value *to be displayed* at that pixel is (0.5, 0, 0).

Both are possible.

Pre-multiplied color

In the latter case, (0.5, 0, 0, 0.5), we say the rgb values have "pre-multiplied" by α

$$(r, g, b, \alpha) = (\alpha R, \alpha G, \alpha B, \alpha)$$

RGB is the color that is computed when rendering e.g. with Blinn-Phong or glColor().

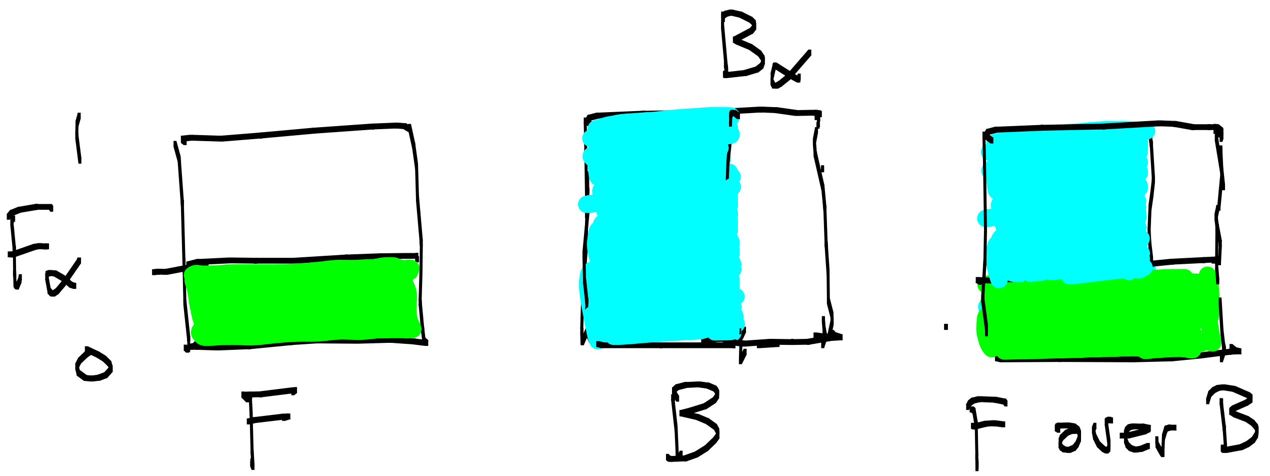
The α is given in the definition of the surface material or in glColor() as in our early example with cyan and yellow triangles.

[ASIDE: Note the similarly to homogeneous coordinates. e.g. (w x, w y, w z, w) represents the 3D point (x, y, z).]

Given $F_{rgb\alpha}$, $B_{rgb\alpha}$,

how do we define $(\,F \text{ over } B\,)_{rgb\alpha}$?

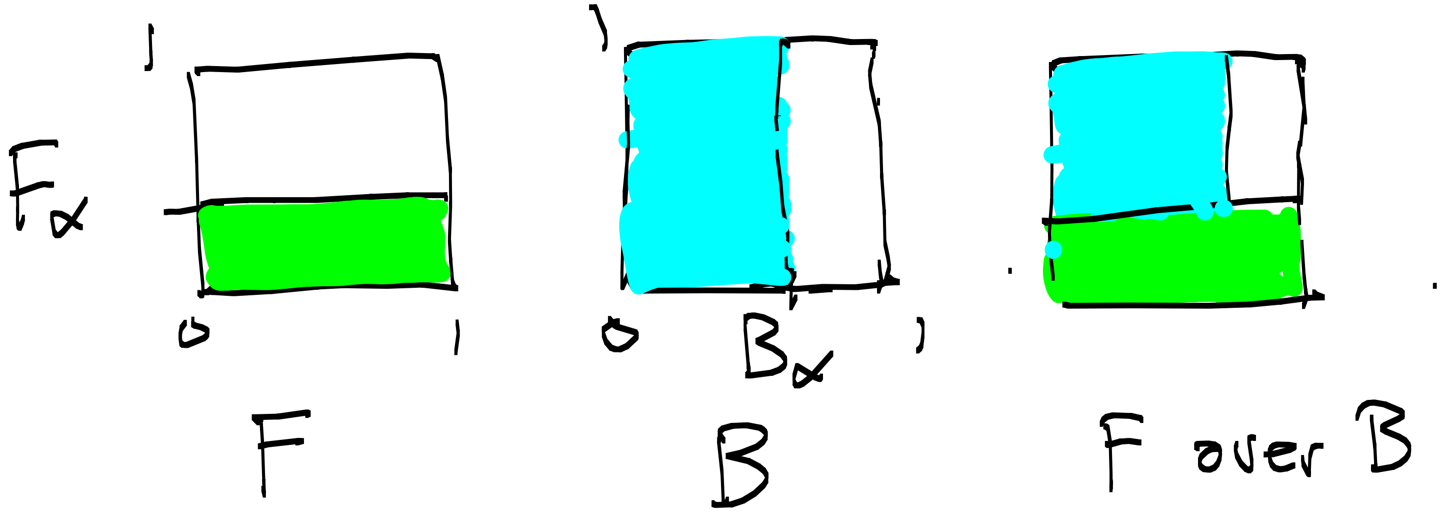
As we argued earlier: assume the geometry below within a pixel. This gives us the formula below for the alpha value of the resulting layer, at each pixel.



$$(\,F \text{ over } B\,)_\alpha = F_\alpha + (1 - F_\alpha)\, B_\alpha$$

If we use pre-multiplied color values,
then we get the same formula for the rgb values:

$$(F \text{ over } B)_{rgb} = F_{rgb} + (1 - F_{\alpha}) \, B_{rgb}$$



That is, by definition of "premultiplied", i.e. $X_{rgb} = X_{\alpha} X_{RGB}$,

$$(F \text{ over } B)_{rgb} = F_{\alpha} F_{RGB} + (1 - F_{\alpha}) \, B_{\alpha} B_{RGB}$$

Exercise: if we don't use premultiplied values, then we get a more complicated formula:

$$(\, F \text{ over } B\,)_{RGB} = \frac{F_\alpha \; F_{RGB} + (1 - F_\alpha)\, B_\alpha \, B_{RGB}}{F_\alpha + (1 - F_\alpha)\, B_\alpha}$$

We are distinguishing two representations:

- RGBA surface properties *that you declare in OpenGL*

  Here, material and opacity are declared *independently* (which is preferable from the programmer's perspective).

- pre-multiplied pixel color values, rgba, that are written in the image buffer

We are distinguishing two representations:

- RGBA surface properties that you declare in OpenGL

  Here, material and opacity are *independent*
  (which is preferable from the programmer's perspective).

  In terms of the graphics pipeline, this is a vertex property.

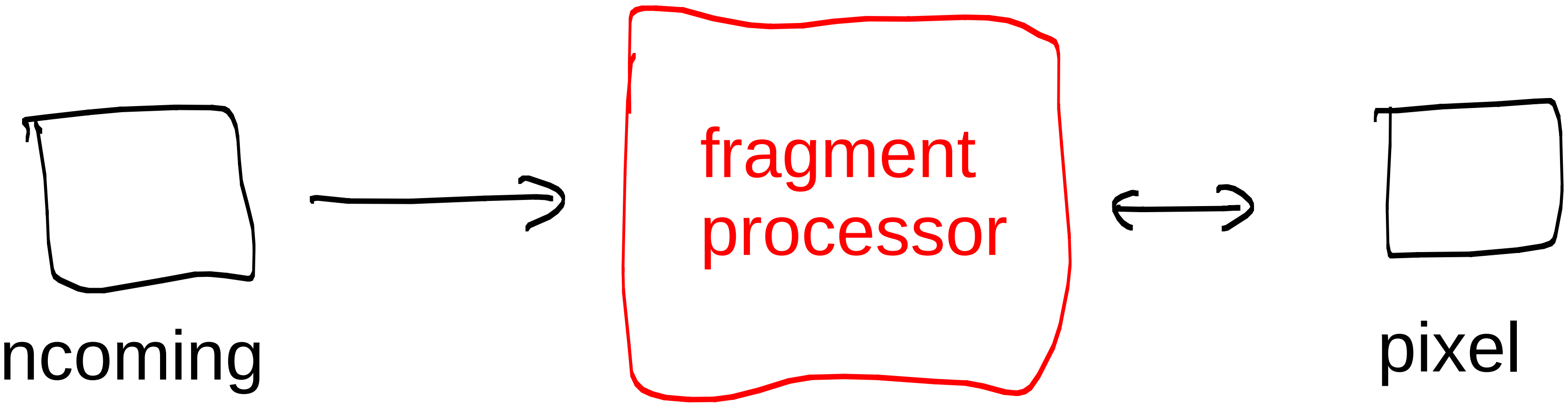
- pre-multiplied pixel color values, rgba, that are written
  in the image buffer

The transformation between the two happens in the fragment shader.

[ADDED: this is oversimplified. It doesn't deal with textures which can also be defined as RGBA.]

OpenGL Blending

Blending must be enabled, else alpha is ignored and incoming fragment is written over the current pixel.



incoming fragment

pixel

r, g, b, α

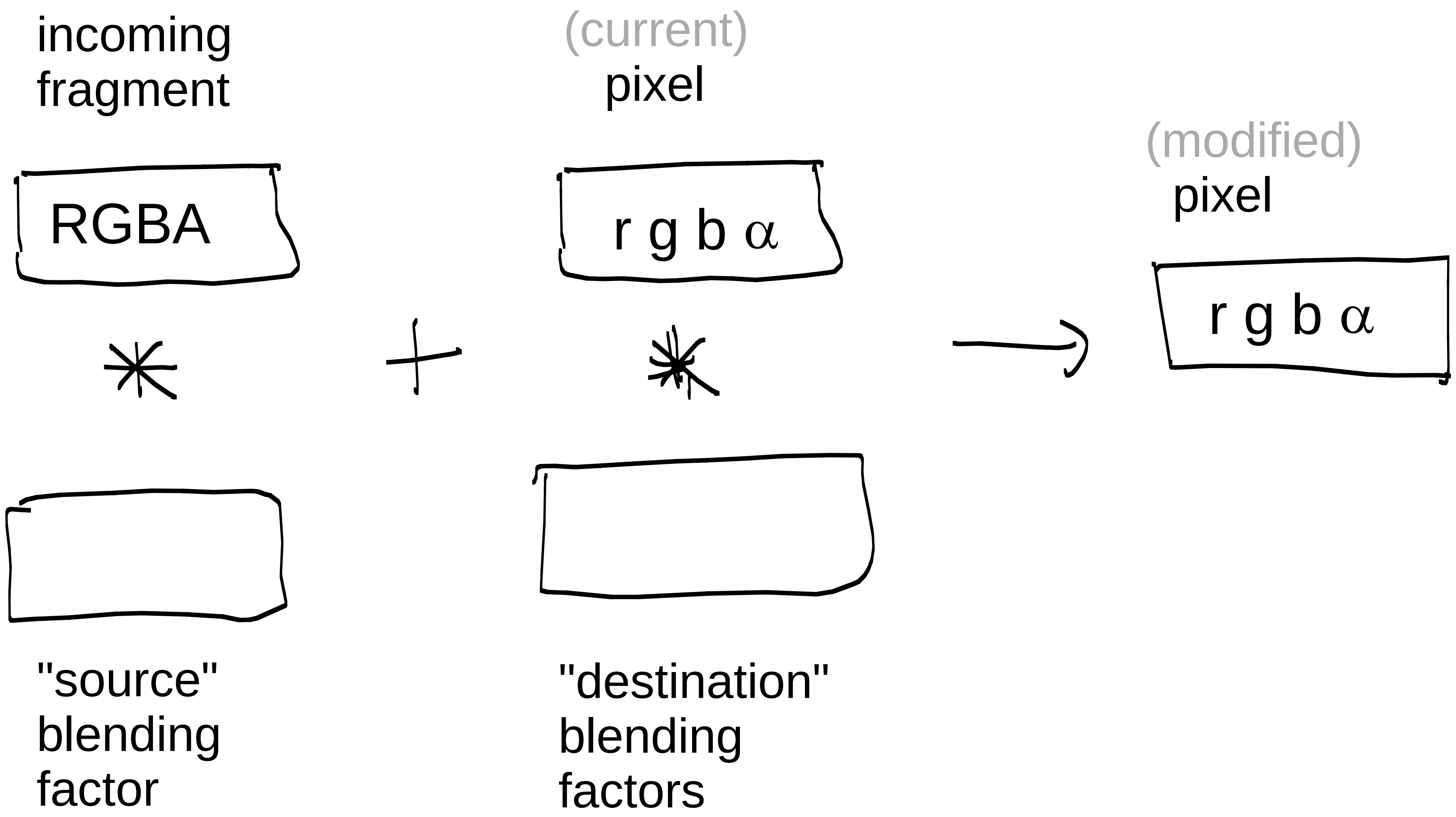
RGB rendered value

n, α, (s, t), z, ...

(pre-multiplied)

The fragment processor takes in fragments and uses them to modify pixels in the frame buffer i.e. image.

The fragment processor takes a fragment, and "blends" it with the current pixel to produce a modified pixel.



Example 1: $(F \text{ over } B)_{rgb\alpha}$

incoming fragment

(current) pixel

(modified) pixel

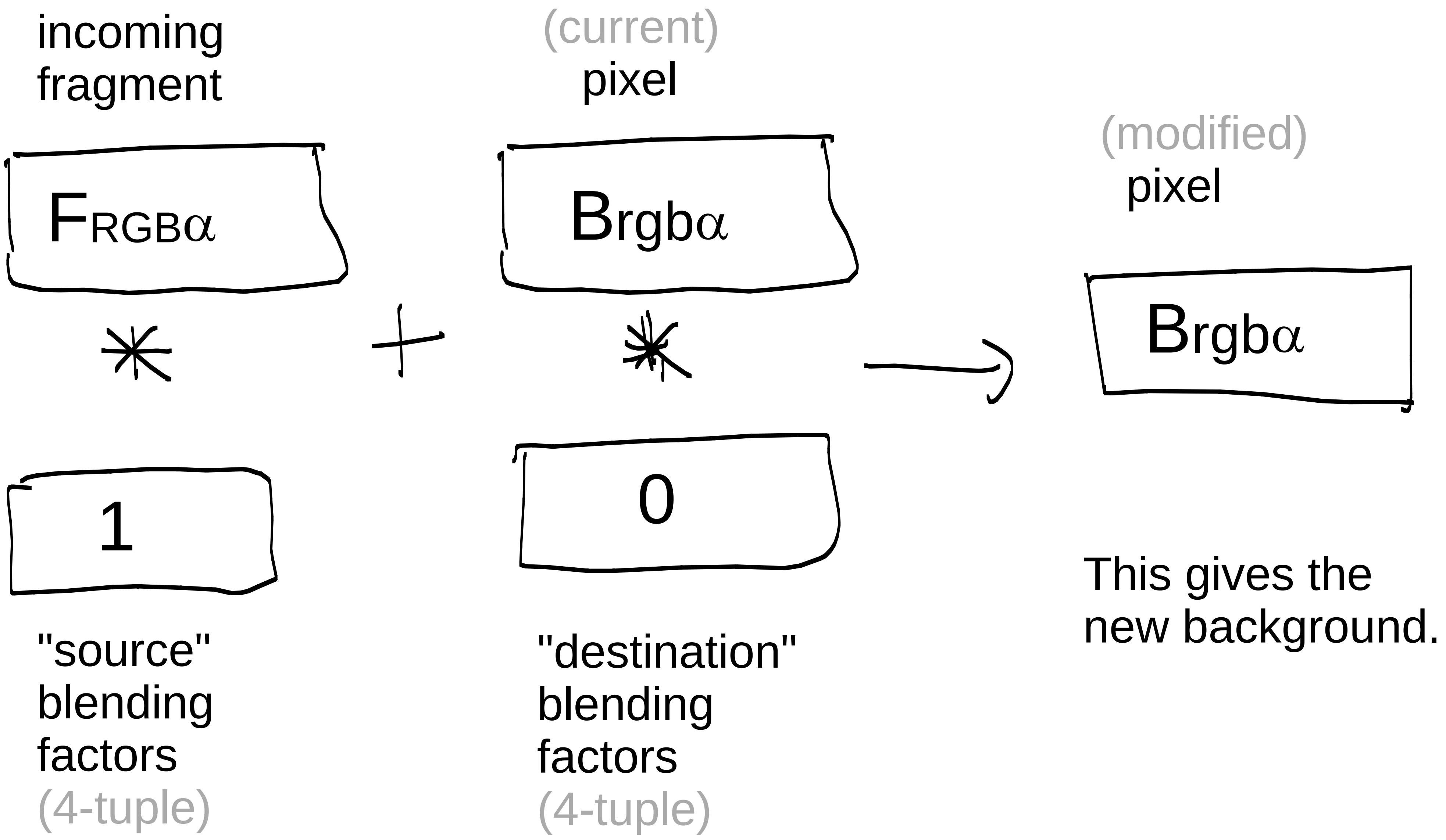
$$\boxed{F_{RGB\alpha}} * \boxed{F_{\alpha}} + \boxed{B_{rgb\alpha}} * \boxed{1 - F_{\alpha}} \longrightarrow \boxed{B_{rgb\alpha}}$$

"source" blending factor (4-tuple)

"destination" blending factors (4-tuple)

This gives the new background.

Example 2: Blending not enabled (default)



This gives the new background.

Classic OpenGL offers several blending functions.

http://www.glprogramming.com/red/chapter06.html#name1

glBlendFunc(source_blending_factor, destination_blending_factor)

For Example 1:

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA).

For Example 2:

glBlendFunc(GL_ONE, GL_ZERO).

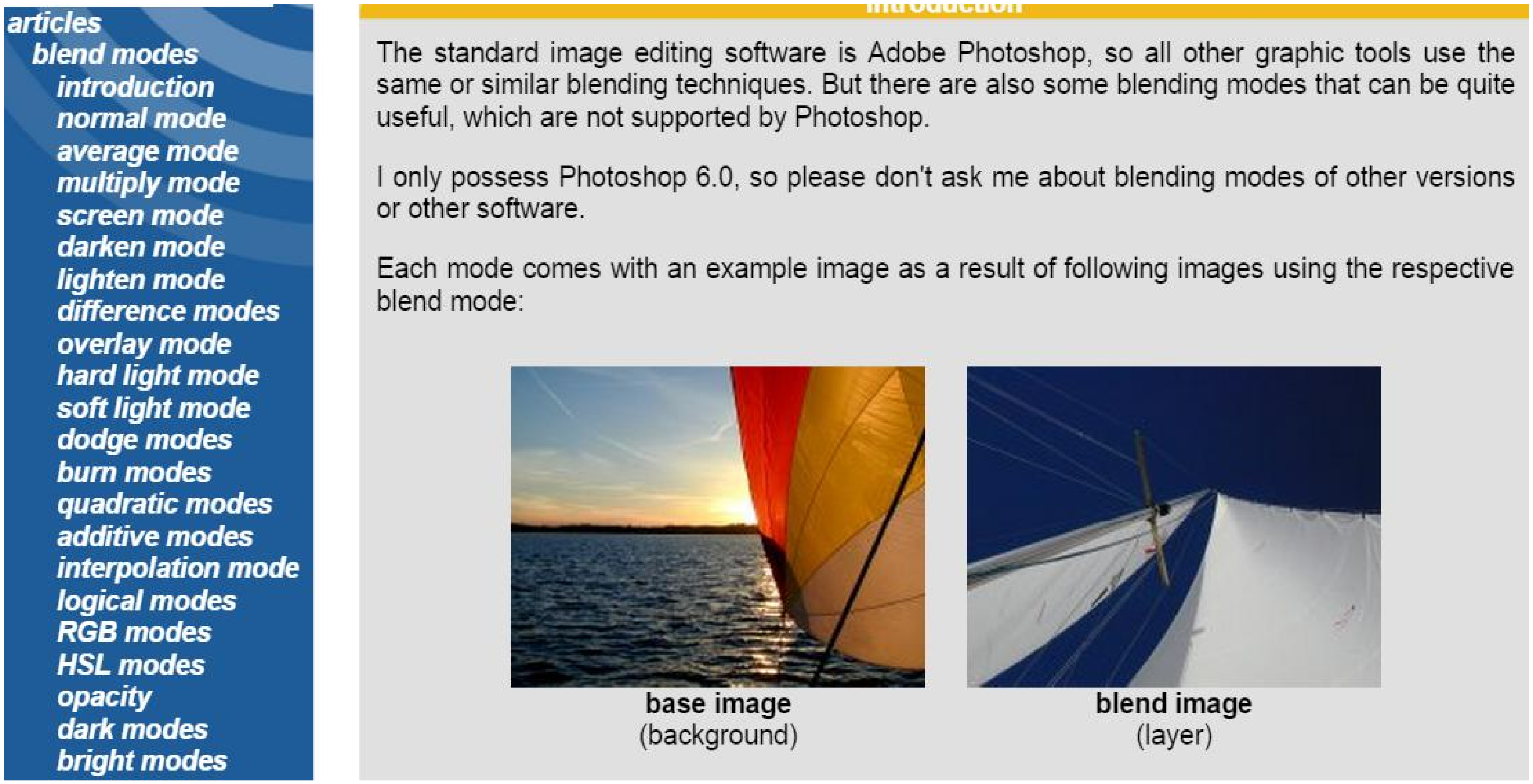
Modern OpenGL allows you to write your own blending functions.

Blending of Image Layers in Adobe Photoshop

http://www.pegtop.net/delphi/articles/blendmodes/intro.htm

- This URL was recommended by the "orange book" OpenGL Shading Language.

See also http://en.wikipedia.org/wiki/Blend_modes



articles
- *blend modes*
  - *introduction*
  - *normal mode*
  - *average mode*
  - *multiply mode*
  - *screen mode*
  - *darken mode*
  - *lighten mode*
  - *difference modes*
  - *overlay mode*
  - *hard light mode*
  - *soft light mode*
  - *dodge modes*
  - *burn modes*
  - *quadratic modes*
  - *additive modes*
  - *interpolation mode*
  - *logical modes*
  - *RGB modes*
  - *HSL modes*
  - *opacity*
  - *dark modes*
  - *bright modes*

The standard image editing software is Adobe Photoshop, so all other graphic tools use the same or similar blending techniques. But there are also some blending modes that can be quite useful, which are not supported by Photoshop.

I only possess Photoshop 6.0, so please don't ask me about blending modes of other versions or other software.

Each mode comes with an example image as a result of following images using the respective blend mode:

base image
(background)

blend image
(layer)

B   F

lecture 20

Image Compositing

- Chroma keying
- alpha
- F over B
- OpenGL blending
- Chroma keying revisited: "pulling a matte"

"Pulling a matte" (image processing)

(alpha channel = a "matte", binary alpha channel = a "mask")

We are given $(F \text{ over } B)_{rgb\alpha}$ and maybe something else.

We would like to

- compute $F_{rgb\alpha}$

- given a new new background B',
  compute $(F \text{ over } B')_{rgb\alpha}$

Alpha estimation using computer vision

Use one image only !

Exercise: Show you have 7 unknown variables at each pixel (but only 3 knowns, namely RGB).

Method:

Assume: F and B have non-overlapping different distributions of colors in 3D color space.

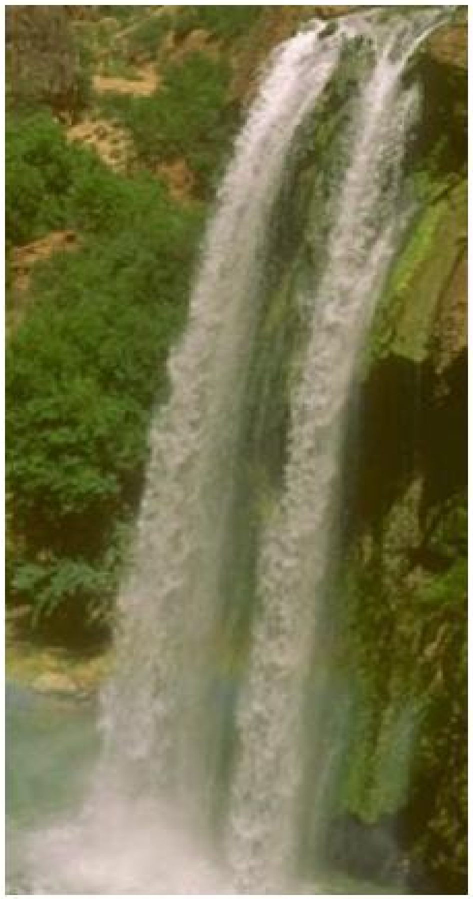
Allowed: user marks by hand regions that that are B and other regions that are in F (and regions that may be in either).
This partitions the image pixels inot three regions, called a **"tri-map".**

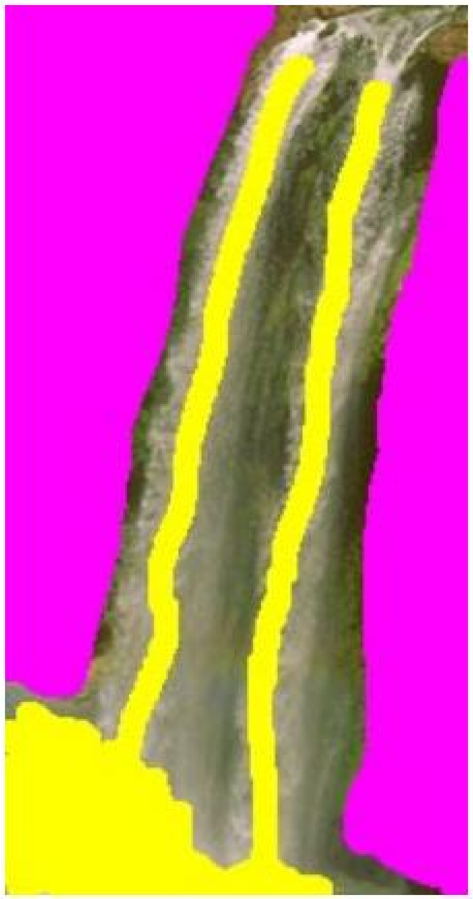
[Ruzon and Tomasi, 2000]

original

"tri-map"

composited result on new background (matte not given)





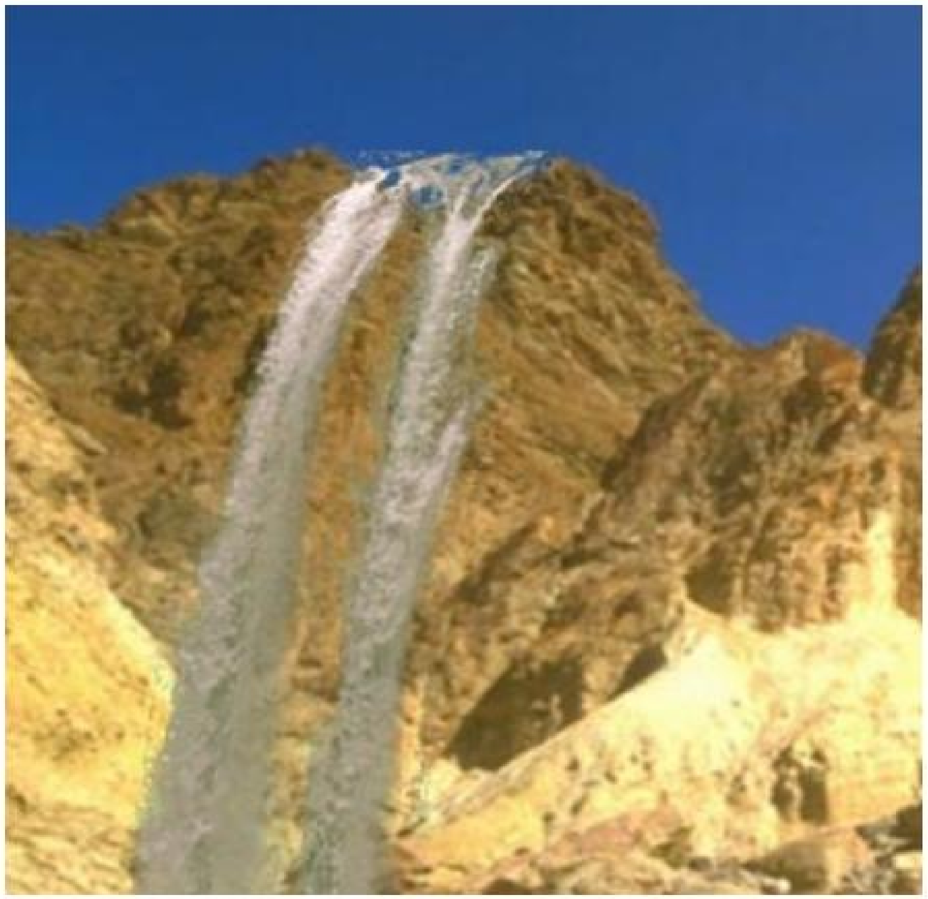
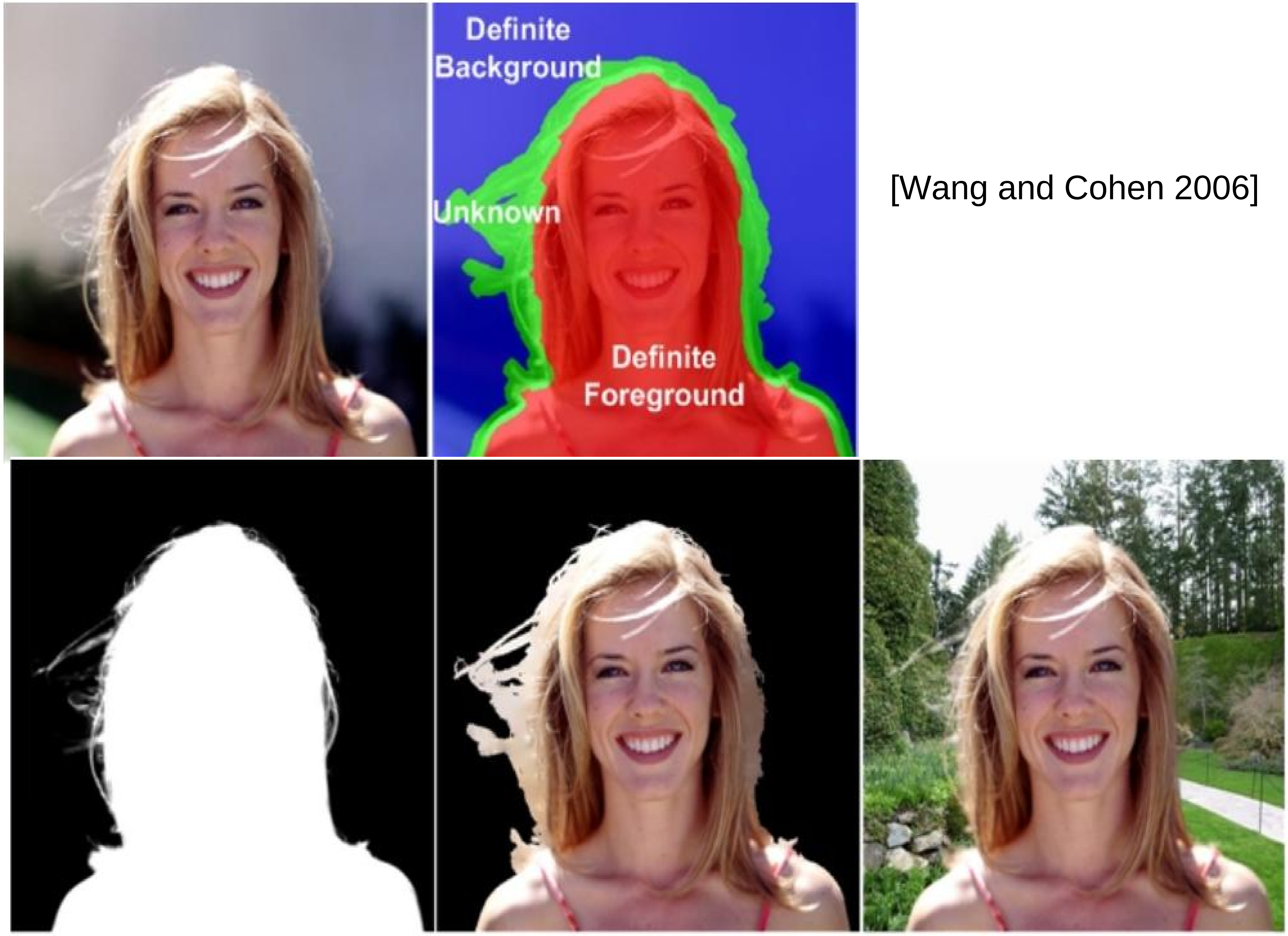


Figure 11. A waterfall is transported to arid Death Valley.

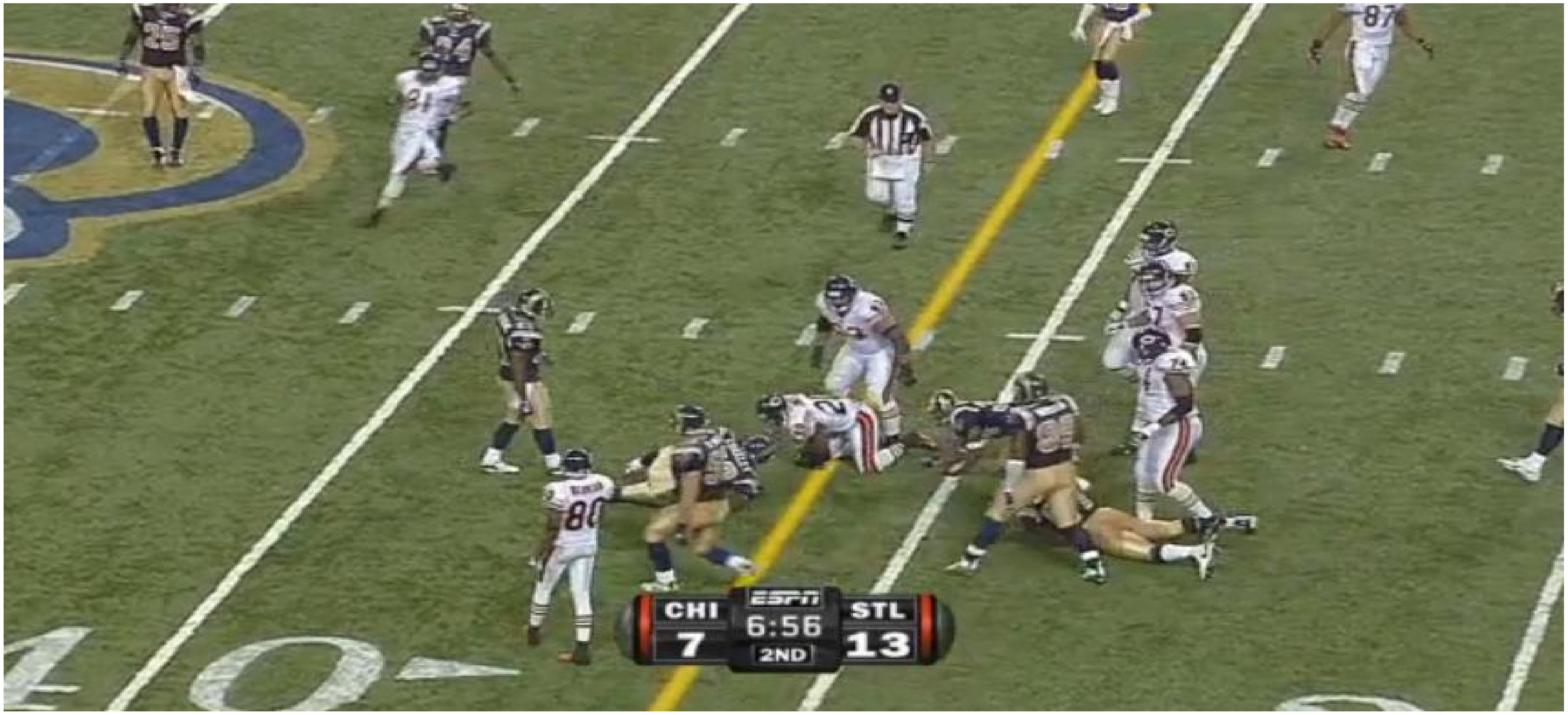


[Wang and Cohen 2006]

A Related Application: "1st and Ten"

http://www.sportvision.com/

http://www.sportvision.com/media/1st-and-ten%E2%84%A2-line-system



Exercise: what must be computed for this to work?