Computer Science 110
The Art & Science of Computer Graphics
Mount Holyoke College
Spring 2010

Topic Notes: Colors and Materials

Colors

We've used Mead's predefined colors and saw a bit about defining our own colors. We will look more at colors and materials next.

Colors in Mead (and in much of computer graphics) use the *RGB* (red-green-blue) color system.

Red, green, and blue are the primary colors of light. Computer monitors (and televisions, etc.) are typically made of lots of red, green, and blue light sources.

In Mead, the values for red, green, and blue are specified in the range 0-1.

The predefined colors are simple `define`s:

```
(define red '(1 0 0))
(define green '(0 1 0))
(define blue '(0 0 1))
(define magenta '(1 0 1))
(define cyan '(0 1 1))
(define yellow '(1 1 0))
(define white '(1 1 1))
(define gray '(.5 .5 .5))
(define black '(0 0 0))
```

Other colors we might think about:

```
(define ltYellow '(1 1 .5))
(define purple '(.5 0 .5))
```

There are many programs that can be used to visualize the colors specified by different RGB values. I like this one:

`http://www.cs.rit.edu/~ncs/color/a_spaces.html`

It's not always obvious how to create the color we are looking for in the RGB scheme. An alternate specification is Hue-Saturation-Value (HSV).

Again, we specify a color by three values:

1. *Hue*: a value in the range 0-360 that specifies the angle on a color wheel
   - red is at 0 degrees
   - we pass through orange, yellow, then on to...
   - green at 120 degrees
   - we pass through cyan
   - blue is at 240 degrees
   - we pass through indigo, violet/magenta...
   - and back to red at 360
2. *Saturation*: a value from 0-1 that specifies the amount of the hue color to mix with white
   - when sat=1, color is the hue
   - when sat=0, color is white

   This allows us to make lighter versions of colors.
3. *Value*: a value in the range 0-1 that specified the amount of this color that you "add" to black
   - when val=1, color is hue+sat
   - when val=0, color is black

What does this all mean? Let's consider a few ranges of values:

- HSV=`(0 * *)` gives us a shade of red (ranging from white through the reds, to black)
- HSV=`(* 0 0)` is black (hue is irrelevant)
- HSV=`(* 0 1)` is white (again, hue is irrelevant)

(here, `*` can be replaced by any value in the range 0-1)

And some examples of HSV colors (note that these are not defined by Mead):

```
(define HSVRed '(0 1 1))
(define HSVGreen '(120 1 1))
(define HSVWhite '(0 0 1))
(define HSVWhite '(90 0 0)) ; either one!
(define HSVBlack '(0 0 0))
(define HSVGray '(0 0 .5))
(define HSVOrange '(30 1 1)) ; 30 degrees around
(define HSVBrown '(30 1 0.5)) ; 30 degrees around, darker
```

A neat feature of the HSV representation is that averaging two HSV values will produce an HSV representation of the "average" color.

Mead, however, does not support HSV colors directly. All colors must be in RGB format.

Fortunately, there is a Mead function that will convert an HSV value to its RGB equivalent: `hsv2rgb`

For example, `(hsv2rgb HSVWhite)` returns `(0,0,0)`, the RGB representation of white.

There is no `rgb2hsv` because there are many representations of white, for example. The function cannot be defined uniquely for all inputs.

Materials

We have seen the predefined "plaster" and "plastic" materials provided by Mead:

See Example:
`/home/jteresco/shared/cs110/examples/PlasterAndPlastic`

Next, we'll see how to create our own materials.

To do this, we create a new object of type `Material` and specify its attributes. This `Material` can then be used for objects we add to our scene.

The attributes we can set for our materials:

- `color`: the material's color in RGB. Remember that each component ranges 0-1. Mead will allow values greater than 1 for the components, but this will result in excessively "bright" objects that are probably not what you are going for. If you want a bright material, instead adjust the parameters below.
- `ambient`: a value in the range 0-1 that specifies how much ambient light illuminates the material. Note that this value is usually very small, since larger values will make the object appear to be unnaturally illuminated without regard to light from actual sources. A non-zero value allows objects that are not illuminated by a direct light source or reflection of light sources off of other objects, to take on their color.

  The default `ambient` value is 0.1.

  Note: unlike in real life, where the ambient light which is reflected by an object can illuminate other objects, POVray (and hence, Mead) does not model ambient light in this way.
- `diffuse`: a value in the range 0-1 that specifies how much of the incident light is scattered using the native color of the object. A value of 0 means that an object's color comes only from ambient light, specularity, transparency, and reflection. A value of 1 (for an opaque object) means that all incident light of colors reflected by the surface is reflected in a scattered fashion.

  The default `diffuse` value is 0.6.

- `roughness`: a value in the range 0 and up, 0 is perfectly smooth, 1 is rough (like a piece of chalk), higher numbers indicate even more rough surfaces. Realistically, values below 0.05 are not useful.

  The default `roughness` is 0.05.

- `specularity`: a value in the range 0-1 that specifies how much incident light from a source is reflected (as if the surface were reflective), leading to "specular highlights". Specular reflection results in, for example, the ability of a glossy paint to reflect some of its incident light.

  The default `specularity` is 0.

- `reflection`: a value in the range 0-1 that specified how much reflection of adjacent objects we see in the surface – 0 is no reflection at all, 1 is a perfect mirror. In most circumstances, values up to about 0.1 are sufficient to add realistic reflectivity to non-mirror objects (such as glass).

  The default `reflection` is 0.

- `transparency`: a value in the range 0-1 that specifies how much light passes through the object – 0 is completely opaque, 1 is a completely transparent object (we won't see it at all).

  The default `transparency` is 0.

- `refraction`: index of refraction for objects with transparency – how much does the light bend when entering and exiting the material. Values greater than 1 will mean light will bend when entering the object.

  The default `refraction` is 1, meaning the light does not bend.

In addition to the above parameters, Mead provides four material "types" that can be used to specify a group of appropriate parameters. We have seen `plaster` and `plastic` materials, but there are also `mirror` and `glass` types.

These are specified in a `Material` definition with the `type`, as we have seen when defining our own plaster colors:

```
(object ltBluePlaster Material
  (type 'plaster)
  (color (hsv2rgb '(240 .5 1)))
)
```

Specifying one of these types sets material attributes as follows:

- `plastic`: `(ambient 0.1) (diffuse 0.5) (roughness 0.05) (specularity 0.7)`
- `plaster`: `(ambient 0.2) (diffuse 0.9) (roughness 1) (specularity 0)`

- `mirror`: `(ambient 0) (diffuse 0.1) (reflection 0.9)`
- `glass`: `(ambient 0) (diffuse 0.1) (transparency 0.8) (refraction 1.5)`

The best way to understand these attributes is to experiment with them.

See Example:
`/home/jteresco/shared/cs110/examples/Materials`

See also the Materials page on the Mead wiki:

`http://cswiki.cs.williams.edu/wiki/index.php/Materials`

This example introduces a new object type: the `Plane`. Several of you have wanted to use such an object, which is an infinite two-dimensional plane that exists (and can be rendered) in three-dimensional space.

The default position of a `Plane` is along the xz-plane. It can be manipulated with transformations just like our familiar basic object types.

See Example:
`/home/jteresco/shared/cs110/examples/DiffuseAndAmbient`

See Example:
`/home/jteresco/shared/cs110/examples/Key`

This example demonstrates more custom objects and adds some shiny materials.

It also includes a new transformation: a *mirroring*. This does exactly what you'd expect: it mirrors the object to which it is applied across the specified plane (in this case, the xy-plane).

Lenses

We can build a lenses out of a refractive material.

A convex lens is like a maginfying glass, a concave lens is like what you'd find in a peephole on a door.

See the Wikipedia page about Lens Optics:

`http://en.wikipedia.org/wiki/Lens_%28optics%29`

See Example:
`/home/jteresco/shared/cs110/examples/Lenses`

In addition to demonstrating refraction, this example introduces a new message we can send to our `image`:

```
(dimensions 1024 768)
```

This changes the size of the generated image. By default, it is 640×480.

Surface Images

We will look in more detail at this later, but as several people have asked, here is how to include a surface image.

For now, we'll just see how to make a poster or a billboard of an image.

See Example:
`/home/jteresco/shared/cs110/examples/Billboard`

The important steps:

1. Create a `Mesh` with a single polygon, onto which your image is to be mapped. This polygon is a special one that defines a surface coordinate system on top of the polygon.

   ```
   (object billboardPic Mesh
         (addUVPoly
           '((-1 -1 0) (-1 1 0) (1 1 0) (-1 1 0))
           '((0 0)     (0 1)    (1 1)   (1 0))
          )
   )
   ```

   This defines a polygon in a mesh in the xy-plane from (-1,-1) to (1,1), but also maps a uv-coordinate system for a unit square onto the polygon with the origin at (-1,-1). More on uv coordinates later.

2. Create a `Material` from an image file.

   ```
   (object clappPicture Material
       (surfaceImage "clapplab.png")
   )
   ```

   In Mead, the image should be in PNG format and should be placed in the `Images` folder off of your home folder.

3. Add the `billboardPic` to the scene (or in this case, to a group that will define a nicer billboard), using the `Material` we've defined. Note that we scale it up from 2×2 to the actual width and height of the image.

This example also introduces a new type of light called a spotlight, which we will look at soon.

A `Material` defined with a `surfaceImage` may also be used in other contexts, and you are welcome to experiment with it. We will see later how to control the mapping of the surface image onto our objects.

Note: you may find that some surface images map better onto objects if you use the `grainImage` message instead of `surfaceImage` when creating your `Material`.

See Example:
`/home/jteresco/shared/cs110/examples/WoodenMarble`