CS98-52 Lecture 7

Wednesday, November 6th, 2019

Today's lecture will discuss strategies for the Scheme recursive art contest, which is an optional strategy contest you can take part in after the Scheme project.

Vector operations

We can represent objects in a three-dimensional space using vectors. We will represent vectors as a three-element tuple.

```
v = (0,0,0)
```

We can define the following basic operations that will help us in our vector operations:

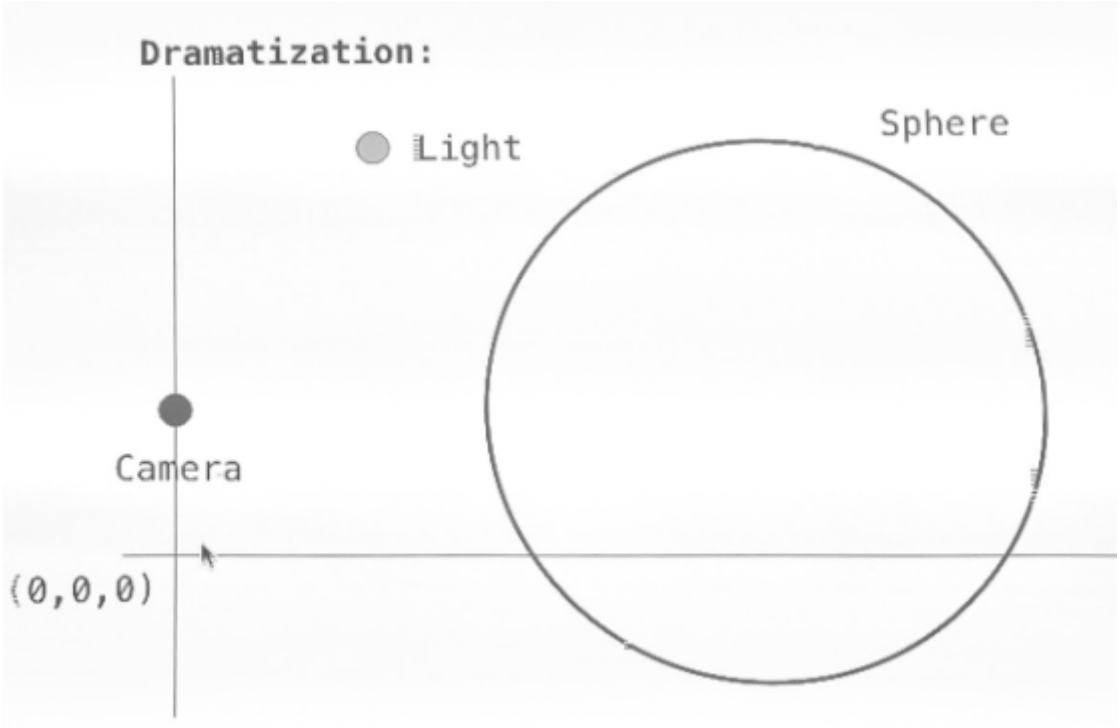
```
def dot(a,b):
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]

def scale(a,k):
    return a[0]*k, a[1]*k, a[2]*k

def normalize(a):
    magntiude = sqrt(dot(a,a))
    return scale(a,1/magnitude)
```

And in today's tutorial, we will define basic locations for the following objects, so we can refer back to them later:

```
camera = (0,1,0)
light = (3,0,0)
black = (0,0,0)
ambient = 0.2
sphere = (1, (0,1,3), (1,1,0))
```

Ray Tracing

Ray tracing is a technique for displaying a 3D scene on a 2D screen by tracing a path through every pixel.

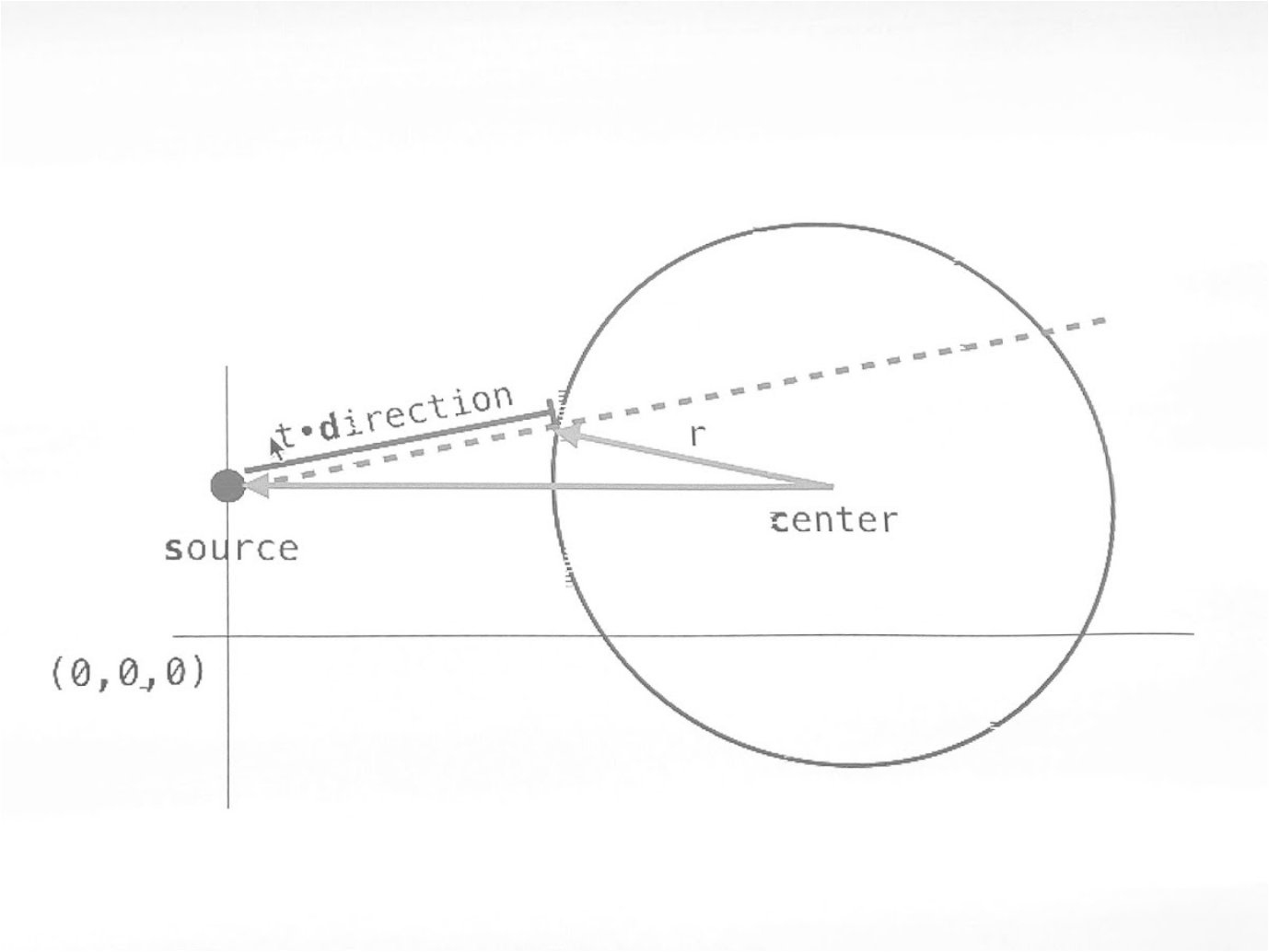


You can compute how every point on the sphere will look in the camera, relative to the light, by using trigonometry and algebra.

How can we compute the distance from a source to a sphere?

We pick the direction we're going from the camera, and figure out the length from the camera to a point on the sphere. We call this line t \bullet dt · d. And with rr being the radius of the sphere:

r^2 = ||s-c+td||^2r2 = ||s − c + td||2 0 = ||td+v||^2 - r^20 = ||td + v||2 − r2
t^2||d||^2 + 2t(v\bullet d) + ||v||^2 - r^2t2||d||2 + 2t(v · d) + ||v||2 − r2
This direction gives us tdtd:



Using this function, we can render the sphere:

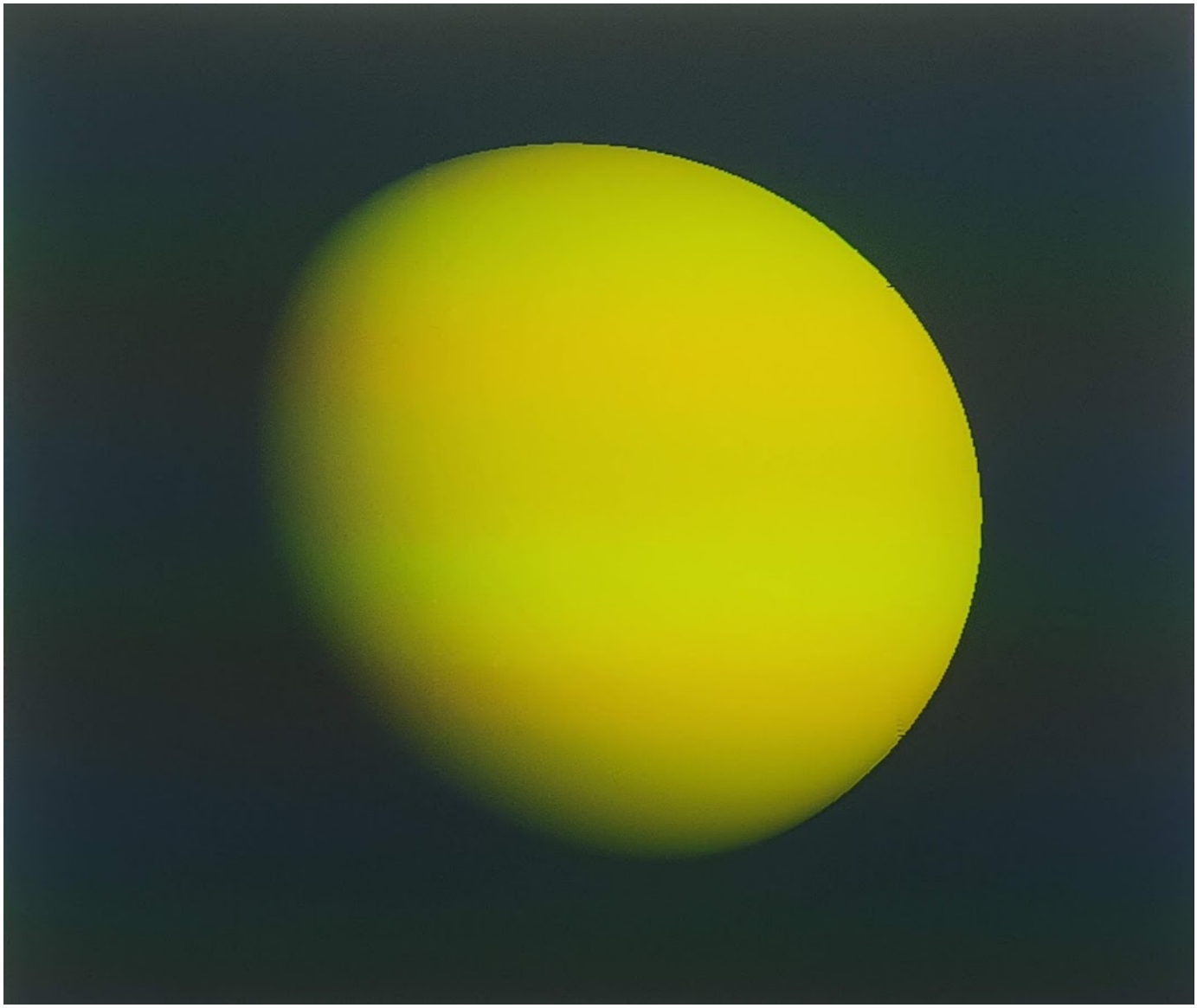
```
def render():
    size = min(width,height):
    for x in range(size):
        for y in range(size):
            direction = normalize((x/size-0.5,y/size-0.5,1))
            color = trace_ray(camera,direction)
            pixel( (x,y), color)

def trace_ray(source, irection):
    distance = intersect(source,direction,sphere)
    if not distance:
        return black
    _, center, color = sphere
    return color

def intersect(source,direction,sphere):
    radius, center, _ = sphere
    v = subtract(source,center)
    b = -dot(v,direction)
    v2, r2 = dot(v,v), radius * radius
    d2 = b*b - v2 + r2
    if d2 > 0:
        for d in (b-sqrt(d2), b+sqrt(2)):
            if d > 0:
```

```
            # This code exists because the line may
            # intersect the circle twice!
                return d
```

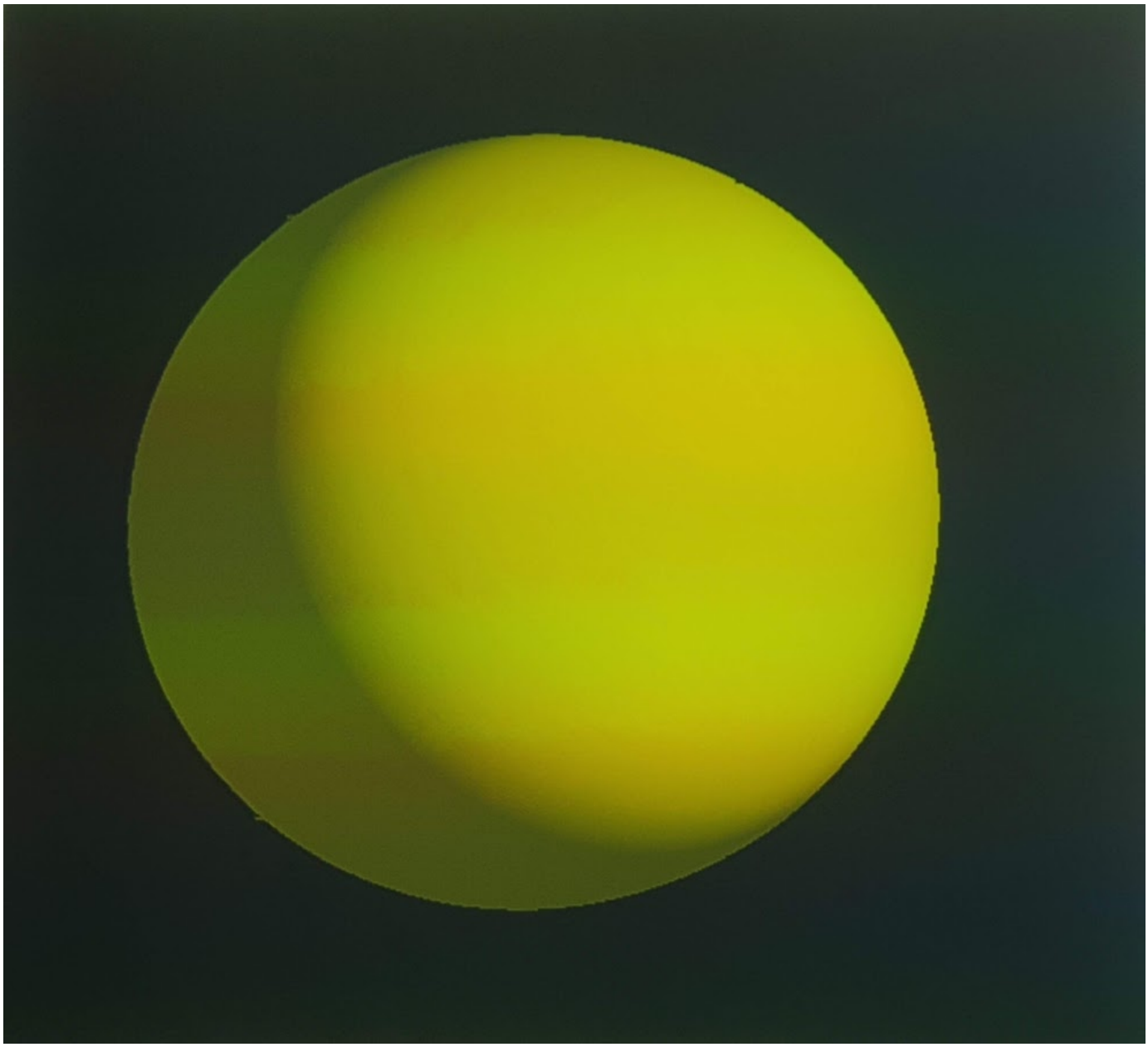
Right now, this code returns a circle by itself, without the cool shading. This happens because we shouldn't just return yellow for any point, but the right amount:

```
def trace_ray(source, direction):
    distance = intersect(source, direction, sphere)
    if not distance:
        return black
    _, center, color = sphere
    surface = add(source, scale,direction,distance))
    return illumination(surface,center,color)

def illumination(surface,center,color):
    to_surface = normalize(subtract(surface,center))
    to_light = normalize(subtract(light,surface))
    intensity = max(0,dot(to_light,to_surface))
    return scale(color,intensity)
```

This code will take a bit to run, but it should return to us a correctly illuminated sphere:



This isn't how a sphere looks! In space, maybe, but on Earth, we almost always have ambient light, so we change the `intensity` variable in `illumination` to our value of ambient light and...



Multiple Spheres

This all is very tough Python, but someone did discover it. You could probably have discovered it yourself, but it would've taken a long time. What you could do, however, is find ways to extend this ray tracing to other functionality, like drawing multiple spheres.

There are a few basic rules you should follow:

- Compute distance to each sphere
- The camera should see the pixel color from the closest sphere

So, we can simply modify our ray tracing function to account for this:

```
def trace_ray(source, irection):
    distances = [intersect(source,direction,sphere) for s in spheres]
    hits = [(d,s) for d,s in zip(distances,spheres) if d]
    if not hits:
```

```
        return black
    _, center, color = sphere
    return color
```

Reflections

Here is the entry that won the recursive art contest a couple years back:

Color is a mixture of the sphere and reflection

The source of a reflection is the surface of the sphere, instead of the original source camera. Pretend you had a camera at the source of the reflection, then mix that color captured with the color of the object reflecting it.

We can do this by picking a depth:

```
def trace_ray(source, direction, depth=4):
    distance = intersect(source, direction, sphere)
    if not distance:
        return black
    _, center, color = sphere
    surface = add(source, scale,direction,distance))
    return illumination(surface,center,color,direction,depth)

def illumination(surface,center,color,direction,depth):
    to_surface = normalize(subtract(surface,center))
    to_light = normalize(subtract(light,surface))
    intensity = max(0,dot(to_light,to_surface))
    direct = scale(color,intensity)
    if depth == 1:
        return direct
    else:
        cosine = dot(direction, to_surface)
        bounce = subtract(direction, scale(to_surface, 2*cosine))
        reflected = trace_ray(surface,bounce,depth-1)
        return mix(direct,reflected,0.5+0.5*intensity**30)
    # intensity**30 is a mathematical trick to account
    # for the fact we can't see reflections in super
    # bright parts of the sphere

def mix(a,b,r):
    return add(scale(a,r),scale(b,1-r))

    def intersect(source,direction,sphere,min_dist=0):
    radius, center, _ = sphere
    v = subtract(source,center)
    b = -dot(v,direction)
    v2, r2 = dot(v,v), radius * radius
    d2 = b*b - v2 + r2
    if d2 > 0:
        for d in (b-sqrt(d2), b+sqrt(2)):
            if d > min_distance:
            # This code exists because you don't want
```

```
            # to see the original sphere in the reflection
                return d
```

The depth is there is because we would never stop if we don't pick a certain level to stop at! Increasing the depth will make the reflections go to a higher level, but also make the code run slower.

> This all is terribly slow because we're using Python's built-in turtle graphics software. There are much faster libraries you could import, but that isn't in keeping with our philosophy in 61A.