Computer Graphics

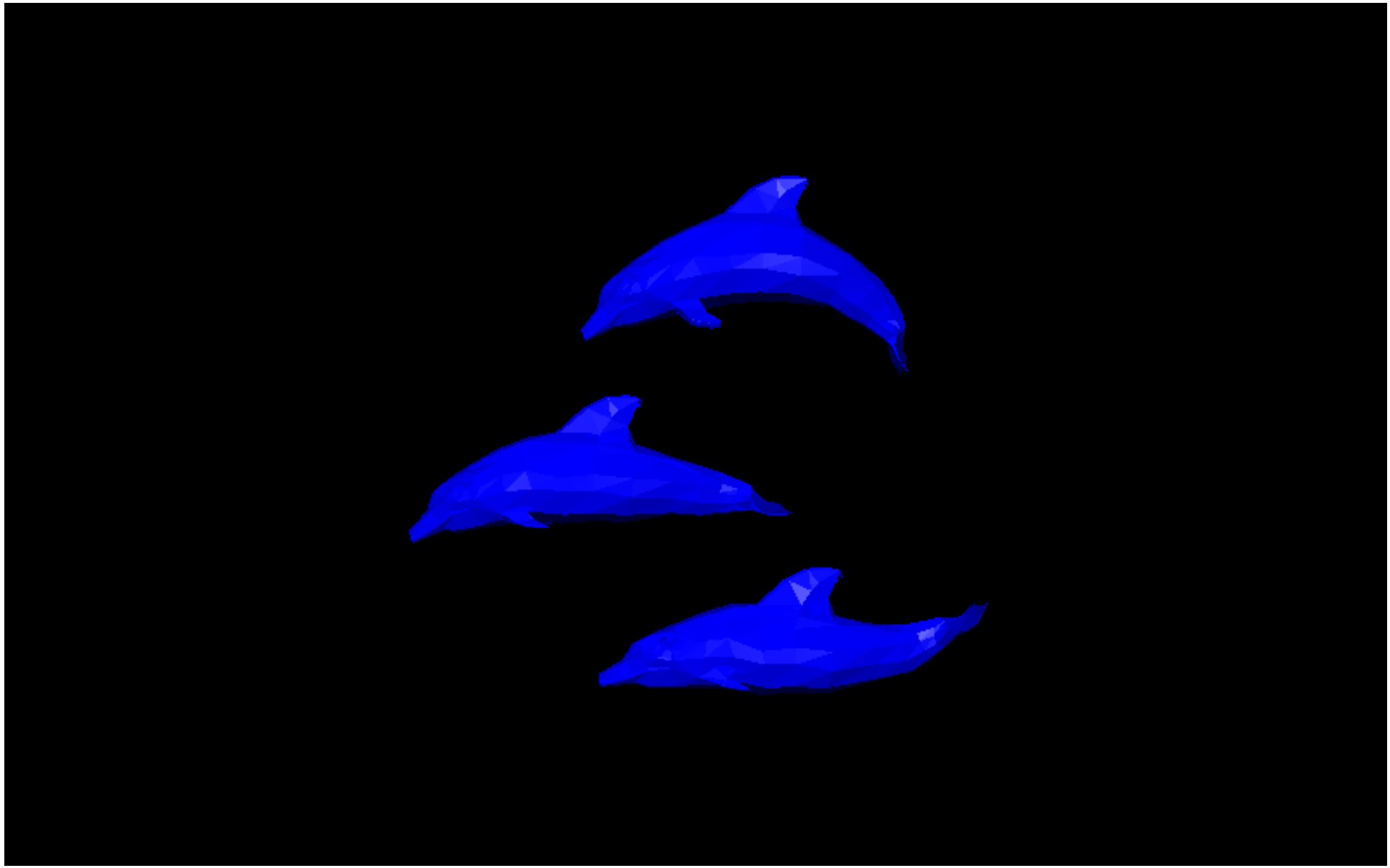
week 6

Roan Kattouw (1770993)
Jan Paul Posma (1775707)

Mesh objects

.obj files already use counter-clockwise triangle orientation: maps directly to our triangle implementation!

Mesh objects

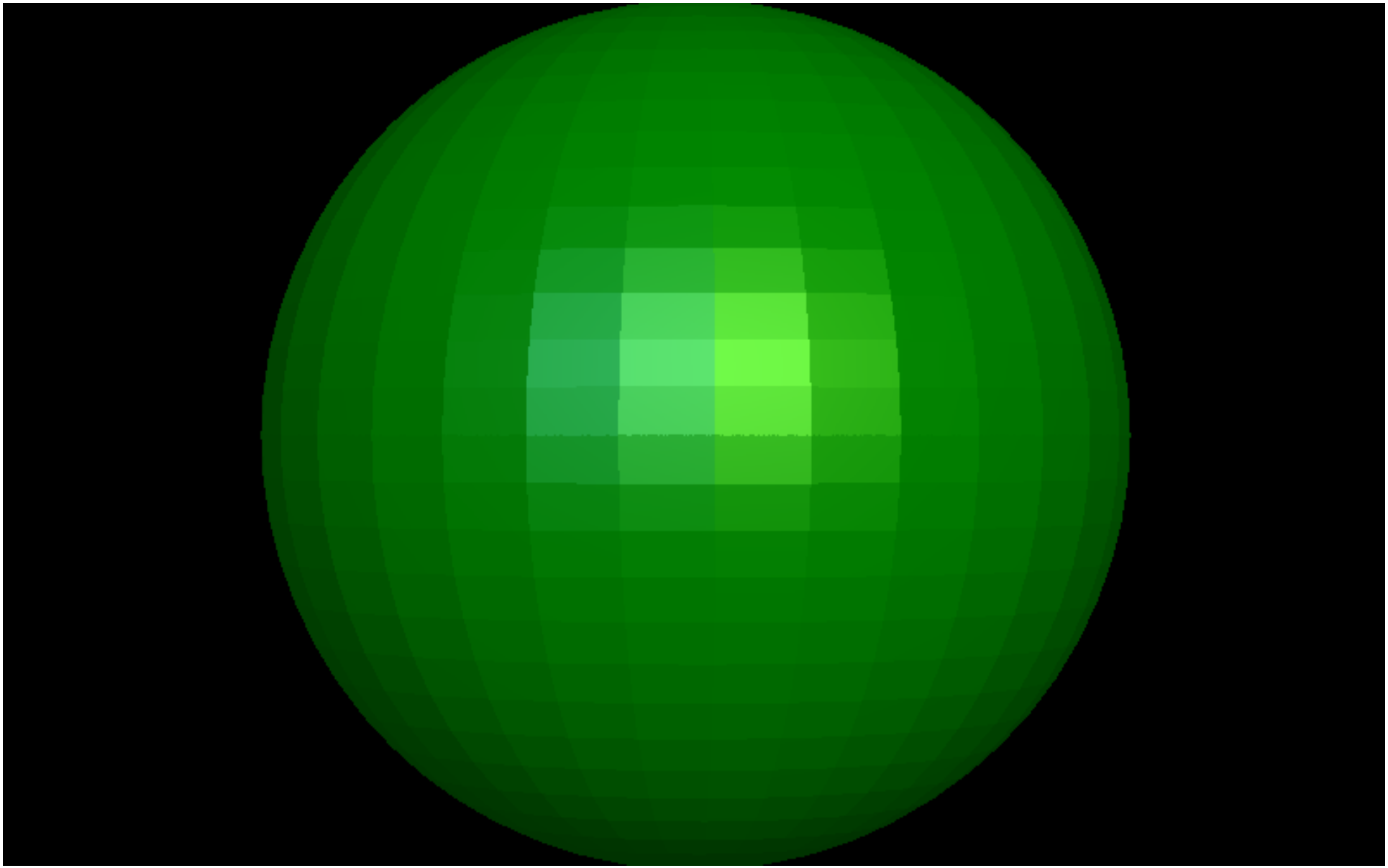


Dolphins!

Mesh objects

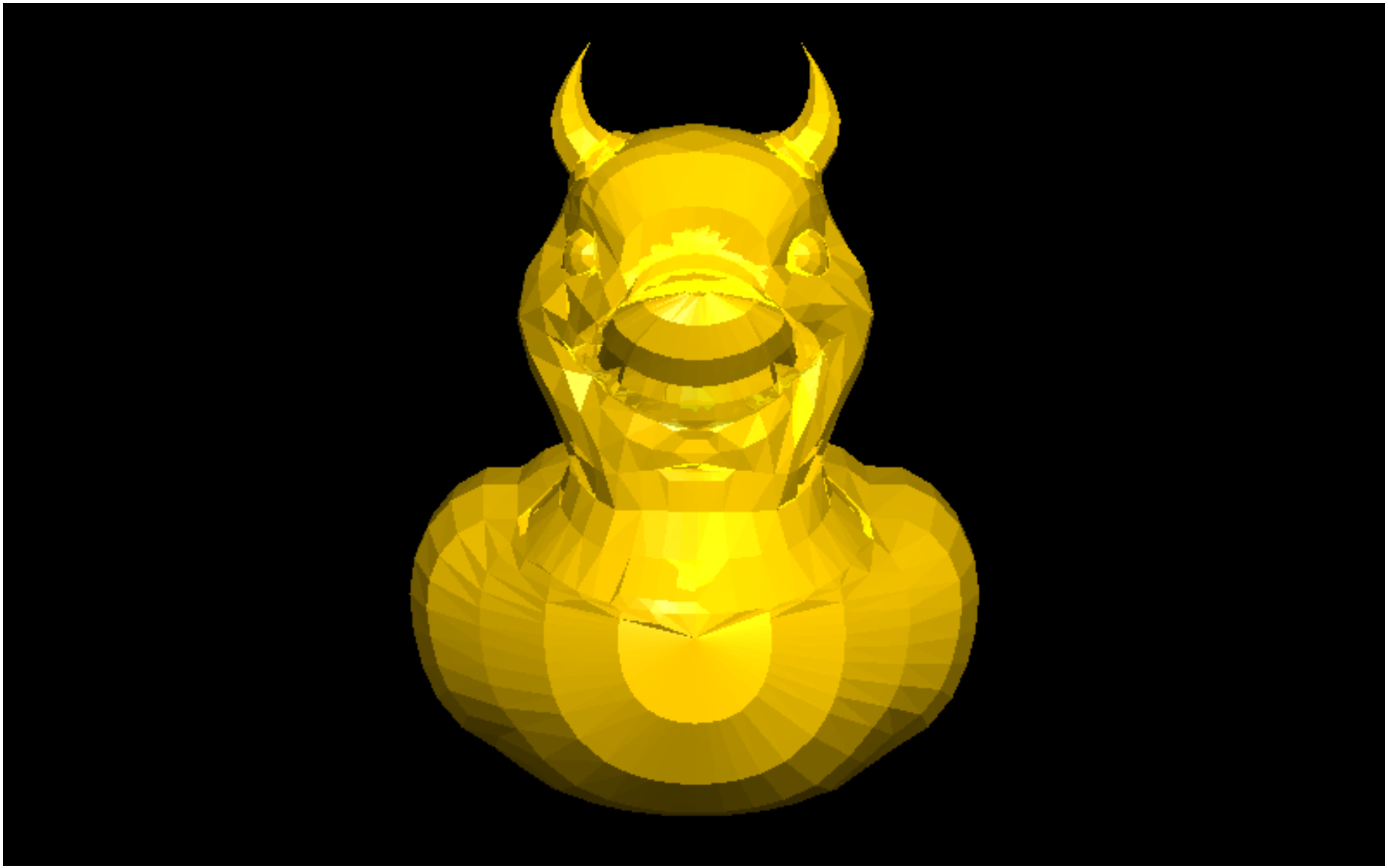
Teddy!

Mesh objects



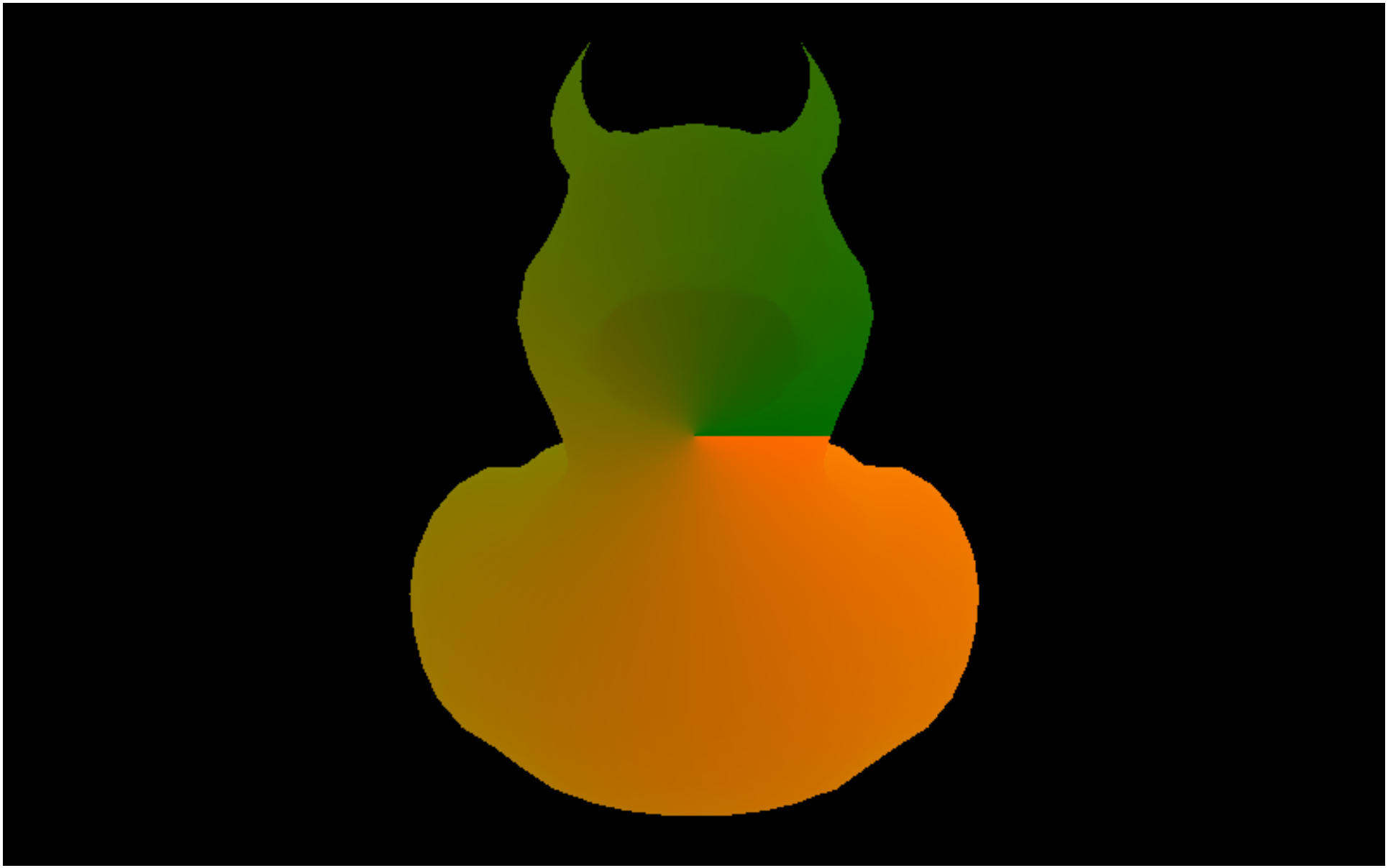
Yay for the ugly and expensive to compute sphere! :D

Mesh objects



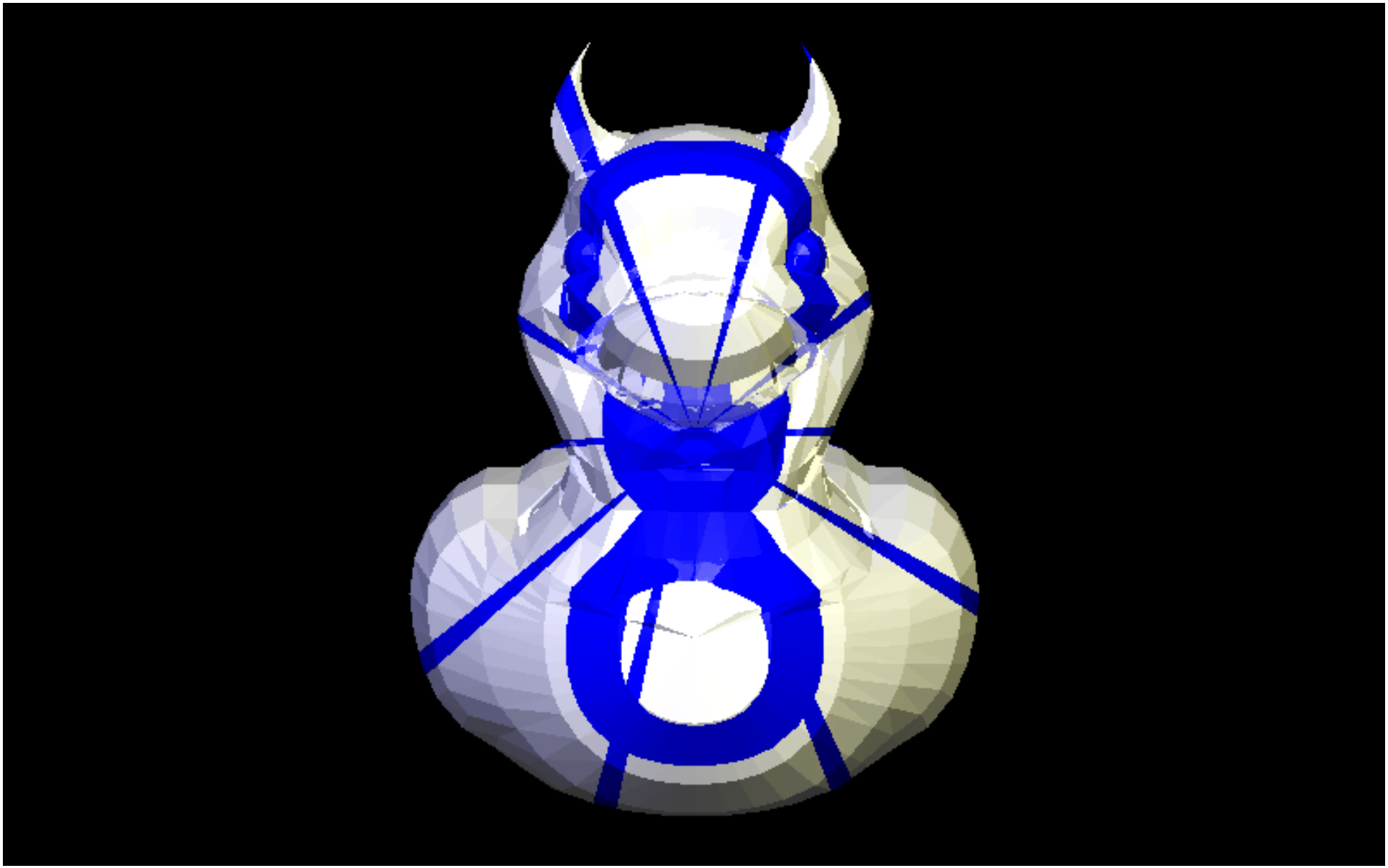
Evil duck!

Mesh objects



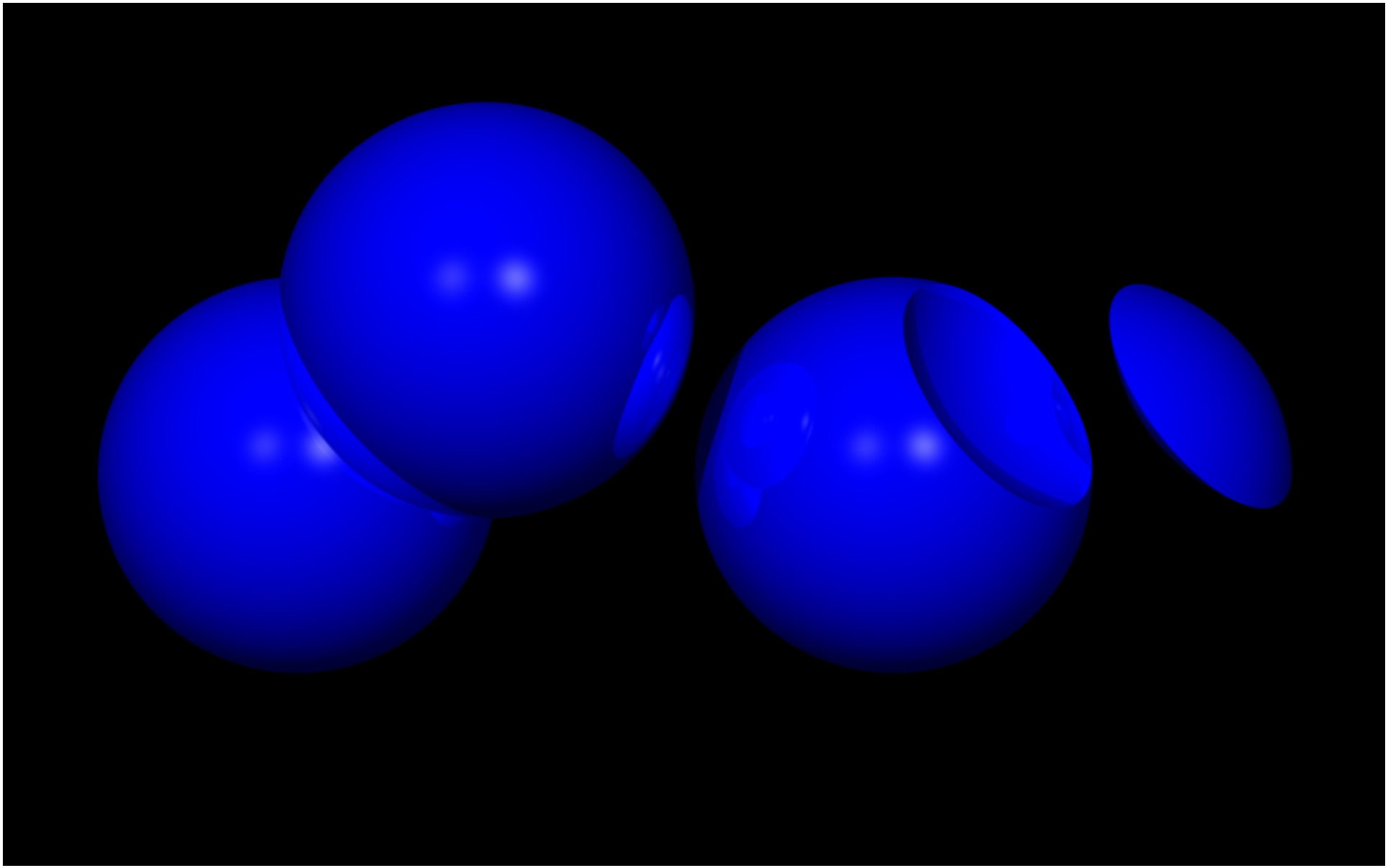
It is trivial to map sphere coordinates to the mesh objects!

Mesh objects



Good stuff ;)

Constructive Solid Geometry



Sweet.

Constructive Solid Geometry

Done by continuing tracing until a certain condition is met:

- union: true
- intersect: inObj1 && inObj2
- difference: inObj1 && !inObj2

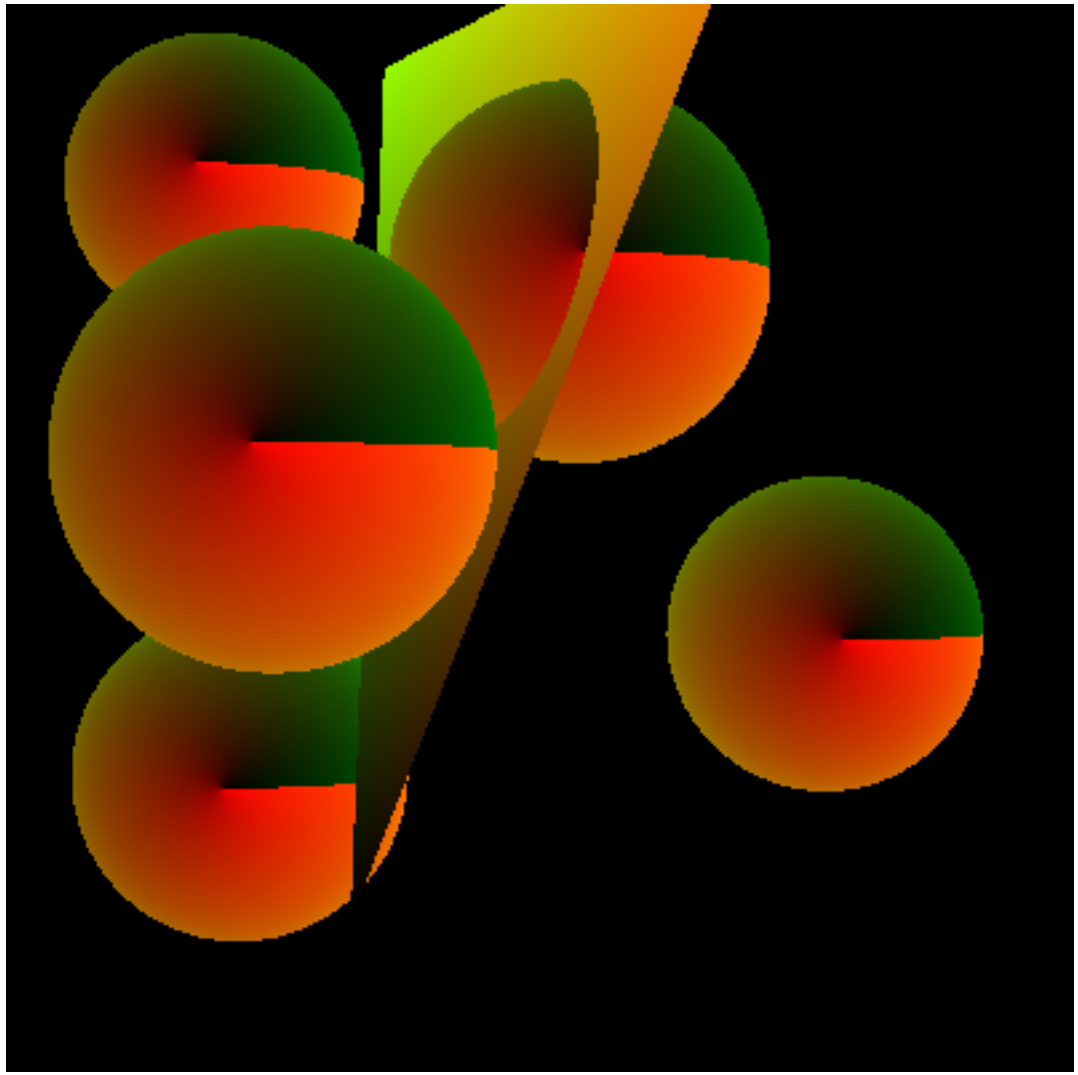
Optimisation: store the intermediate t-values and use the algorithm with those, instead of continuing tracing.

Triangle texture coordinates

Use the dot product of the two vectors p2-p1 and p3-p1 with the point p-p1:

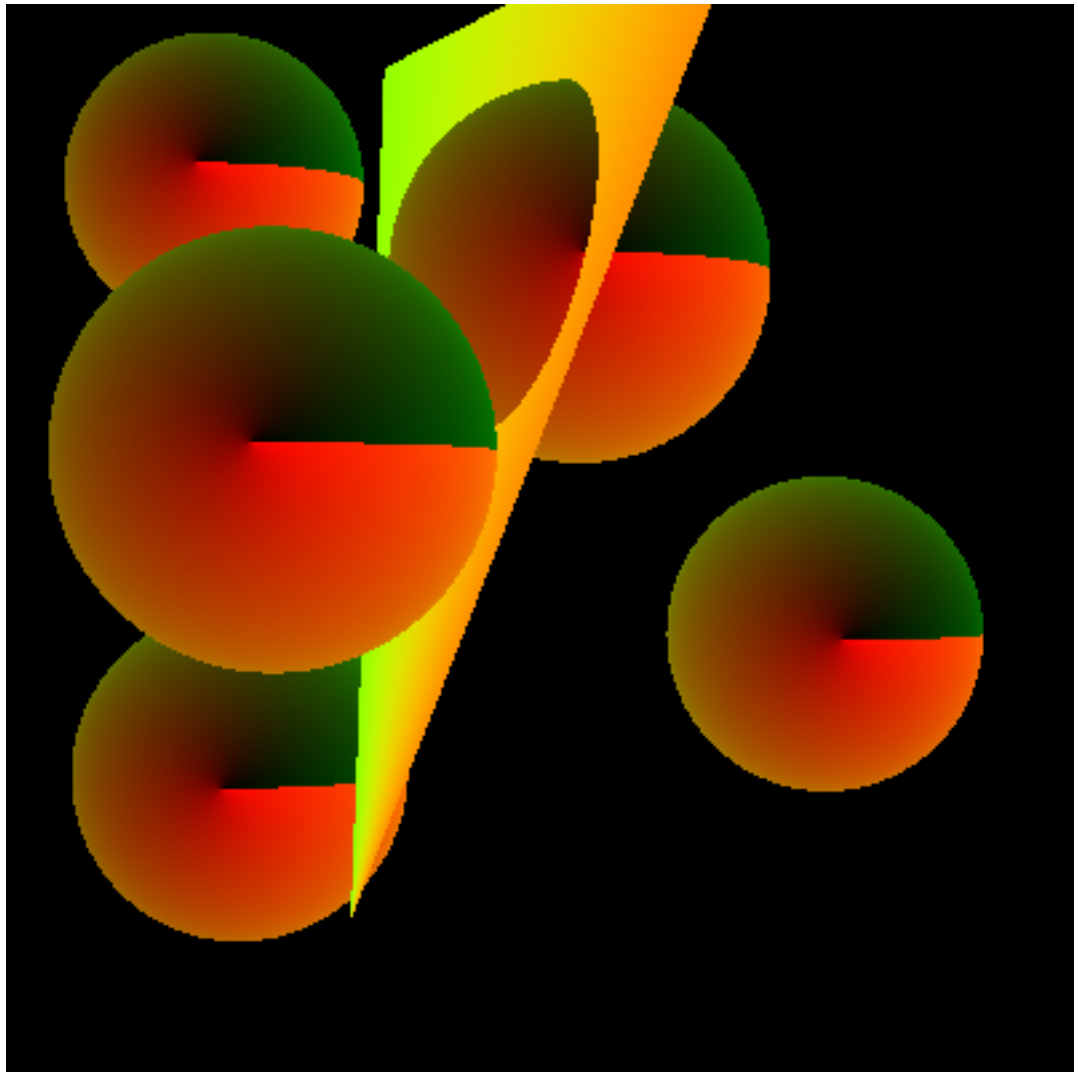
```
Vector d1 = p2-p1;
Vector d2 = p3-p1;
u = d1.dot((p-p1))/d1.length_2();
v = d2.dot((p-p1))/d2.length_2();
```

Triangle texture coordinates



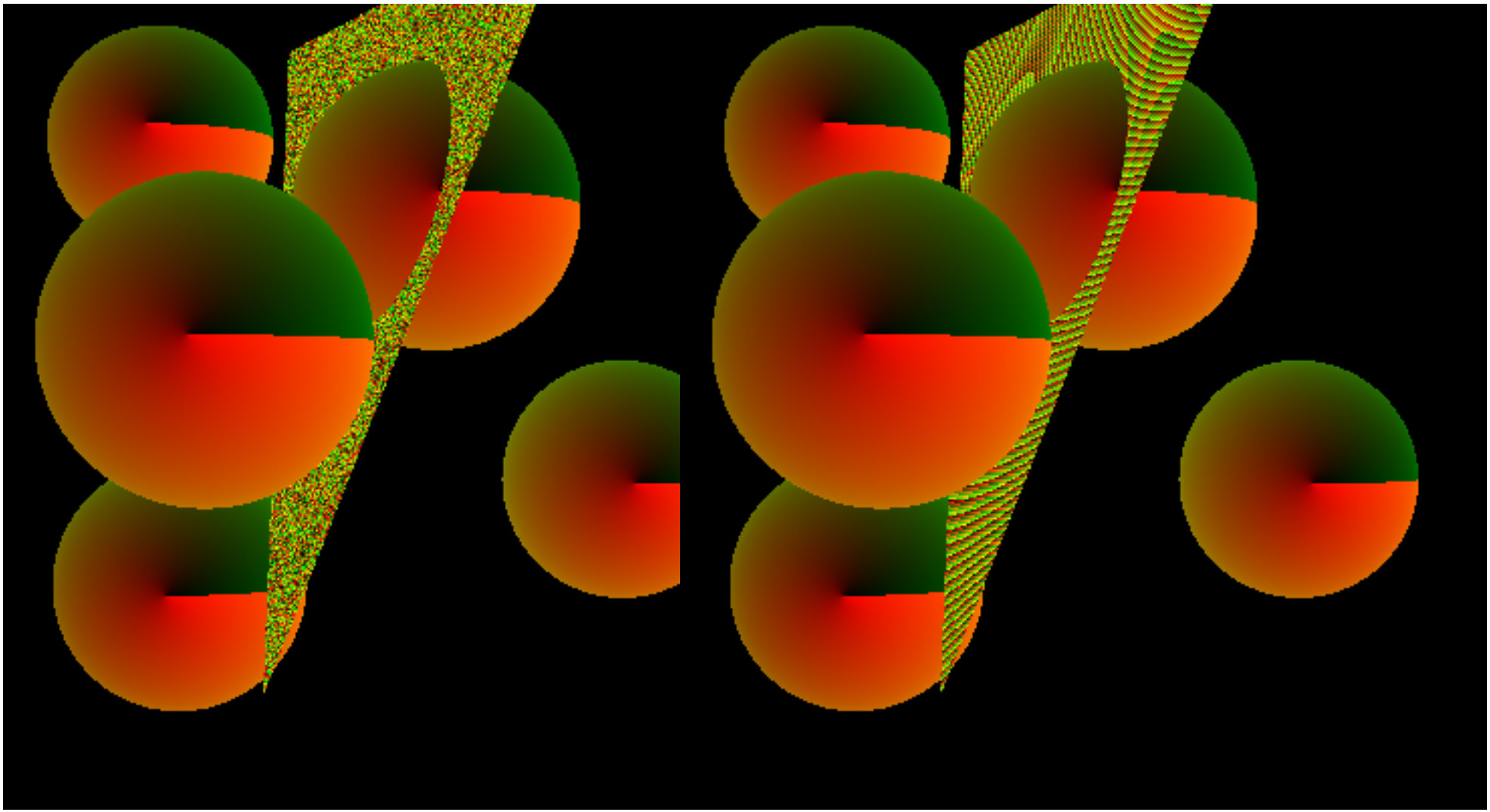
Like this!

Triangle texture coordinates



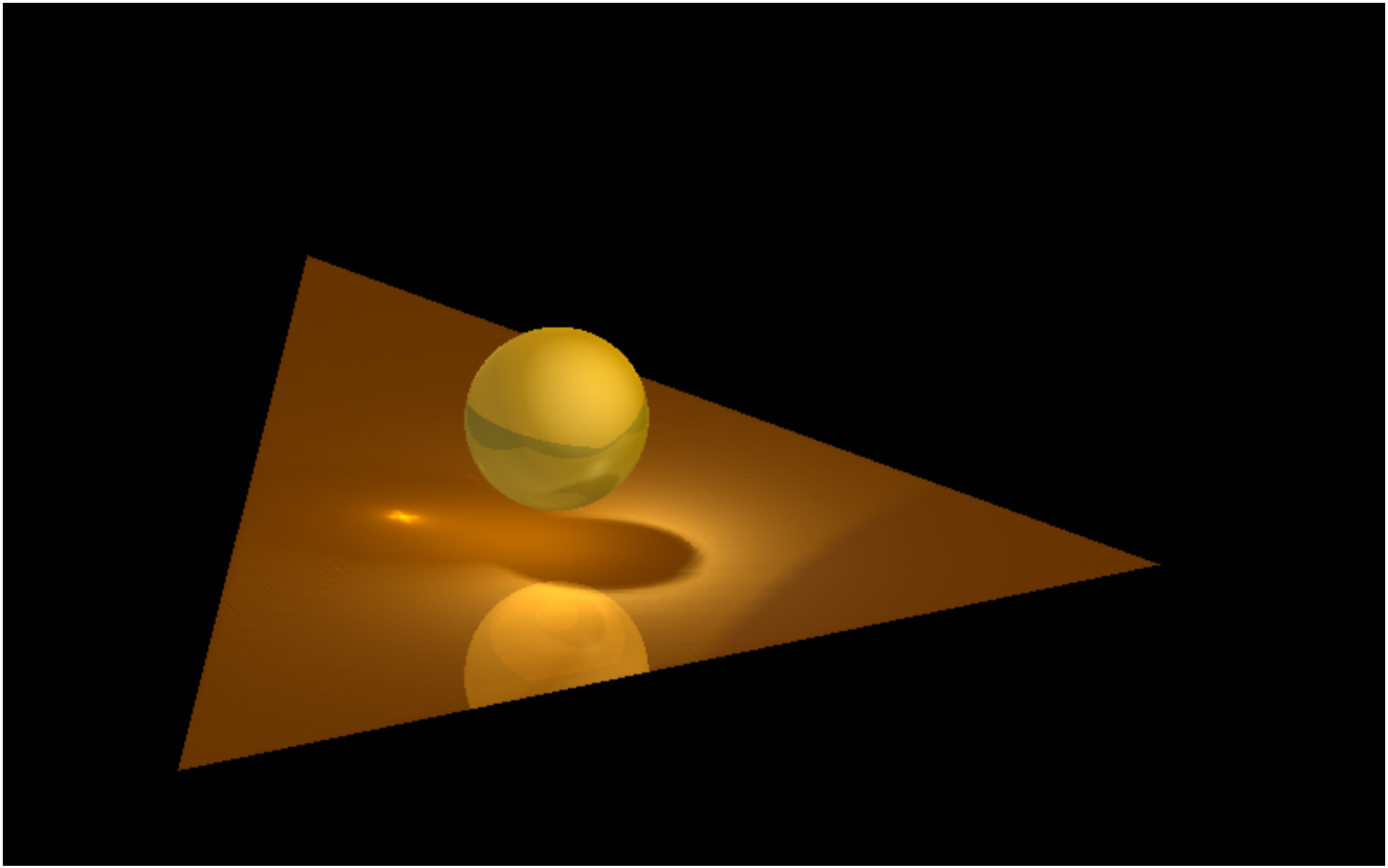
Not this! When using normalised vectors, only the angle is taken into account, not the distance from p1

Triangle texture coordinates



Also not these: incorrect scaling factors: dot product should be divided by the length of a side, squared (as dot product is |A| * |B| * cos theta, and |A| and |B| both have this length as maximum)

More photon tracing optimisations



Problem: why is this a cross and not a circle?

More photon tracing optimisations

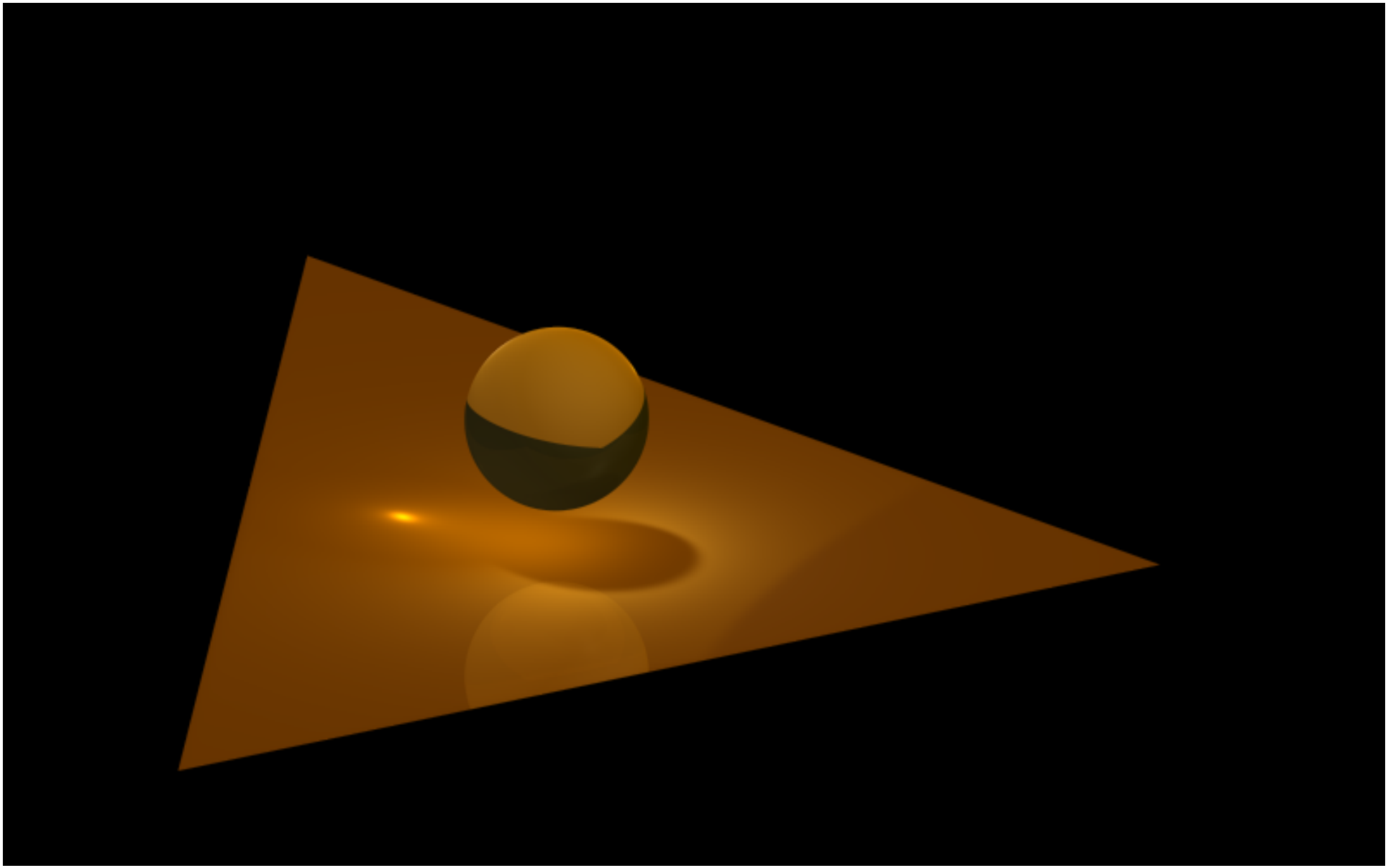
Answer: simple blur function only blurs in x and y directions!

Solution: better blur function, such as Gaussian blur.

Implementation: precompute Gaussian function coefficients, then blur in x direction, then in y direction. (as the function is linearly separable!) Only blur in the range of 3.0*sigma (±99.7%)

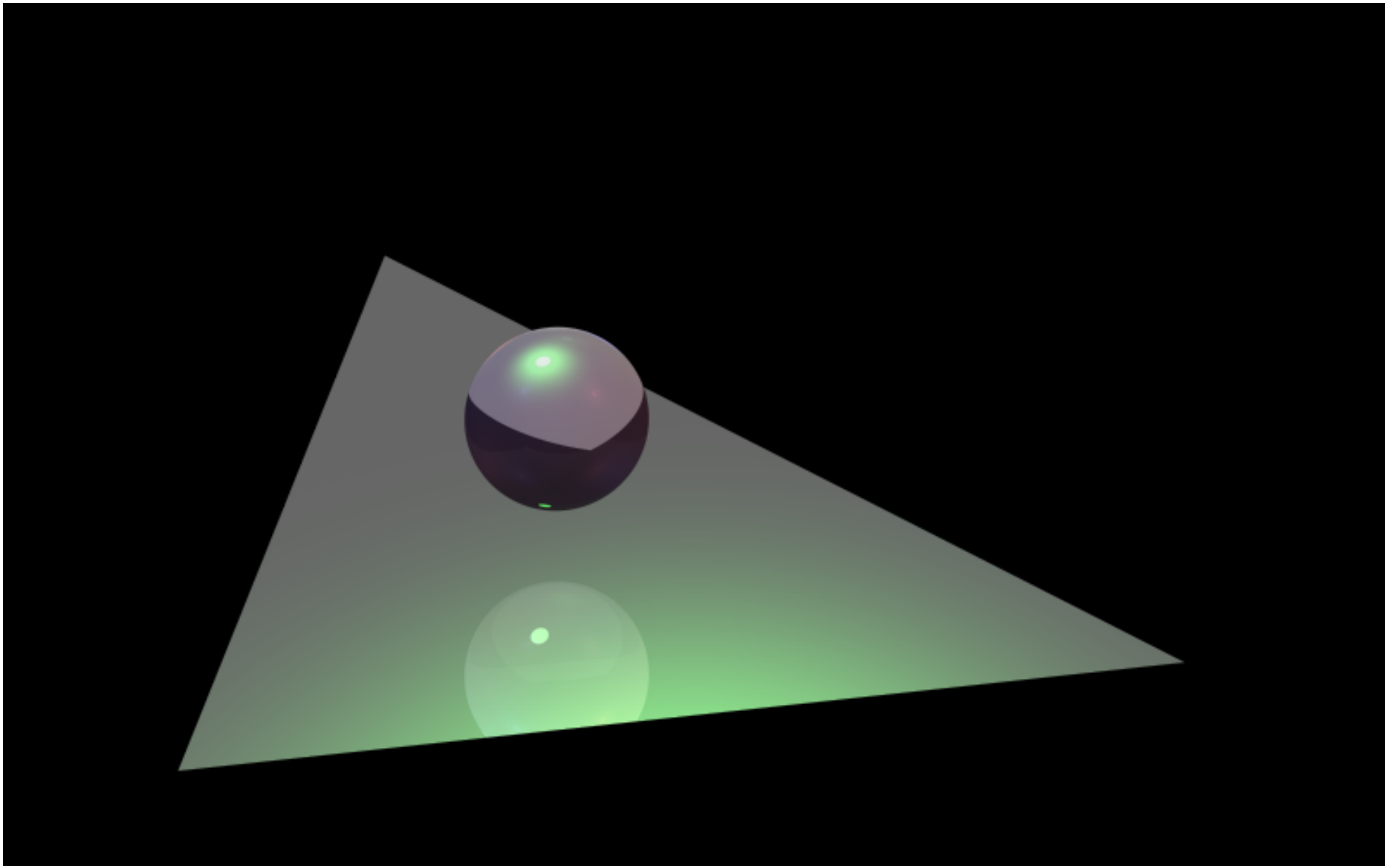
```
// precomputation
double sigma = ((double)radius) / 3.0;
double gauss[radius];
double constant = 1/sqrt(2*M_PI*sigma*sigma);
for (int i = 0; i < radius; i++)
    gauss[i] = constant*exp(-(double)(i*i)/(2.0*sigma*sigma));
```

More photon tracing optimisations



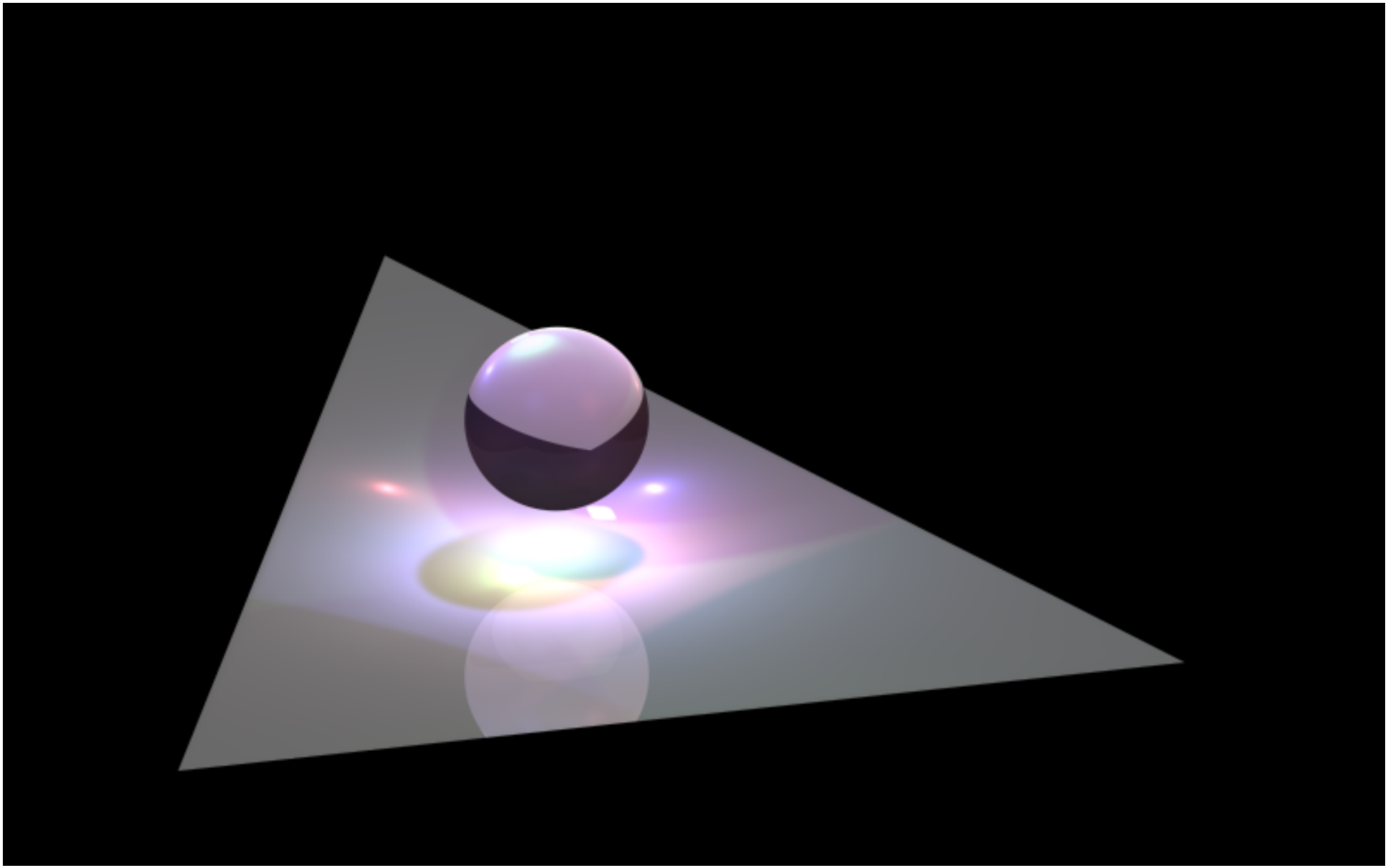
Better! And pretty fast, too.

Visible lights



A visible light (sphere) will actually refract, nice!

Nice images



Photon tracing with multiple light sources :)

New geometry: cylinder

Cylinder equation is very easy if you make some assumptions:

- Describe cylinder with line from A to B and a radius r
- Assume that A=(0, 0, 0) and B=(L, 0, 0)
- P is on the cylinder if Py^2 + Pz^2 = r^2 and 0 <= Px <= L
- Filling in P = O + t*D yields a quadratic equation
- Solve for t using quadratic formula and check x-coordinates of resulting points

We're essentially doing math in another coordinate system, so we need a transformation. We compute a rotation matrix that rotates AB onto the x axis, and use that to transform vectors. Points need to be translated (subtract A) before rotating.

Adding cylinder caps

We want solid cylinders, not hollow ones, so we need to intersect with the caps too. These are circles around A and B in the plane perpendicular to AB.

If you're too lazy to compute intersections between lines and arbitrary planes, like me, you can do the intersection in the transformed coordinate system instead (the planes are x=0 and x=L there). If you also don't want to have to process a variable number of t's between 2 and 4, you can compute plane intersections only for t's with out-of-bounds x values.

Normals for cylinders

The normal is the line from AB to P, except for points on the caps, where it's the line AB (or BA). Forgetting about this special case produces strange-looking mushroom-like caps: