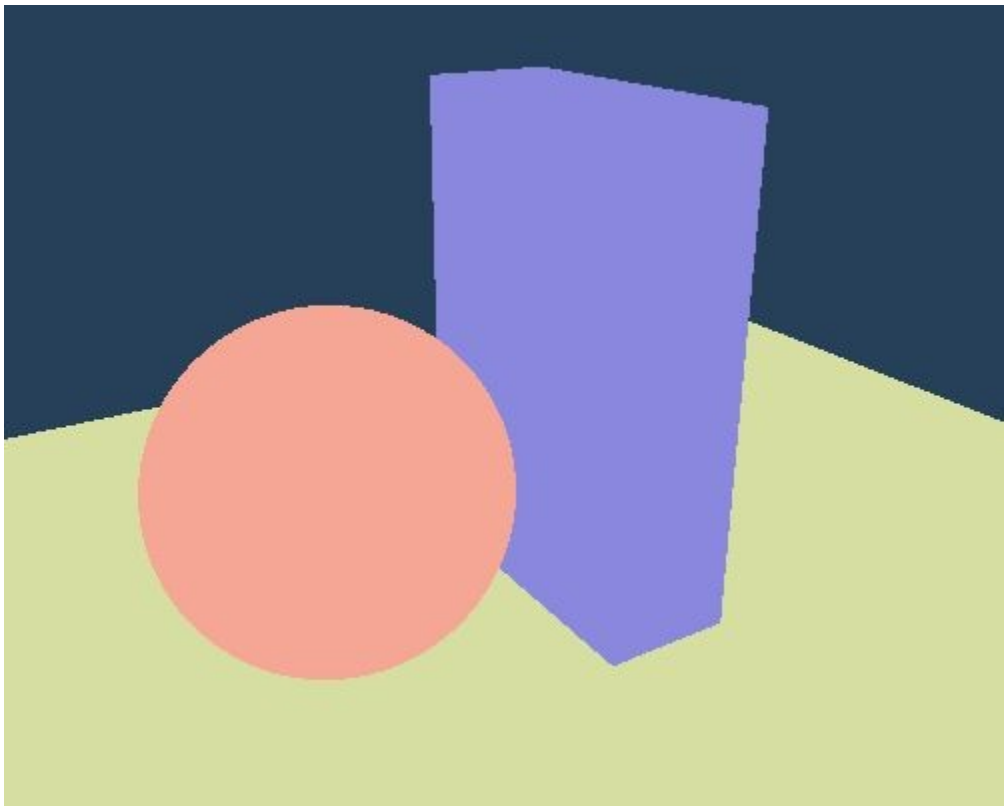
Sean Murphree
Spring, 2011
CS 114

Project 2: Ray Tracing

In this project we were to add to an existing ray tracing program. We were to implement ray/box intersections, diffuse lighting, specular lighting, shadows and reflectivity, the addition of each being a separate stage with it's own results as well as struggles. I was able to implement ray/box intersections, diffuse and specular lighting, but ran out of time to finish shadows.

Ray/Box Intersection:

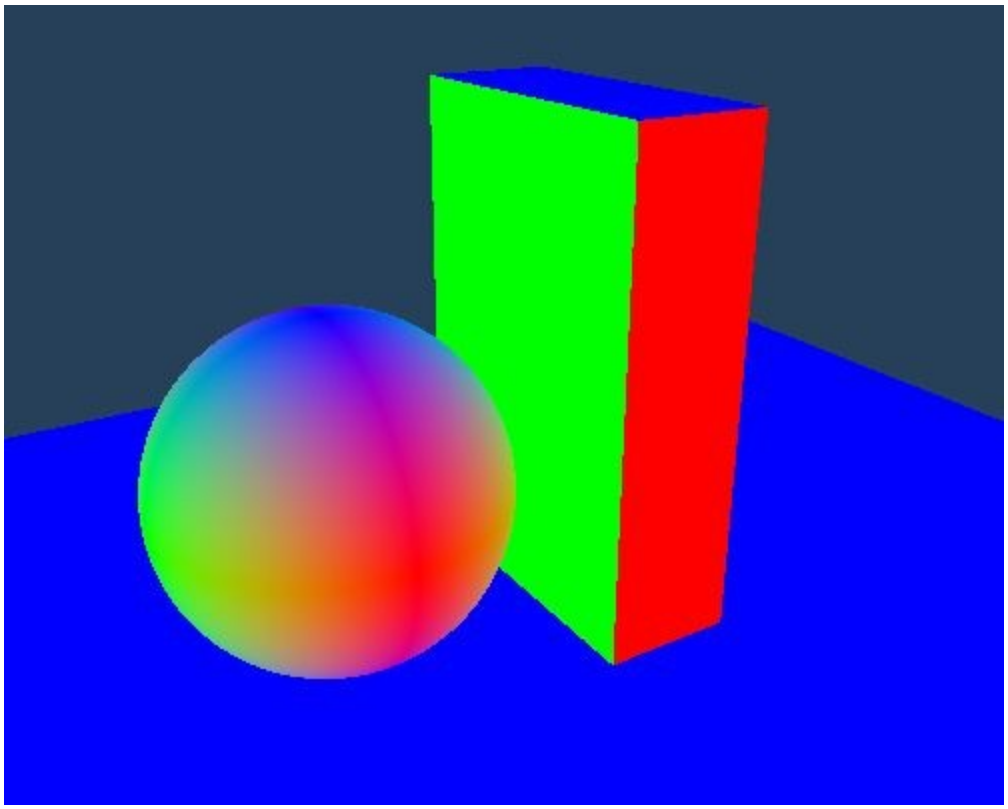
In implementing ray/box intersections, everything was pretty straight forward, but a bug existed with calculating the normals of the intersection point, which didn't show up until implementing diffuse lighting. The result of adding ray/box intersection is seen in the following image:



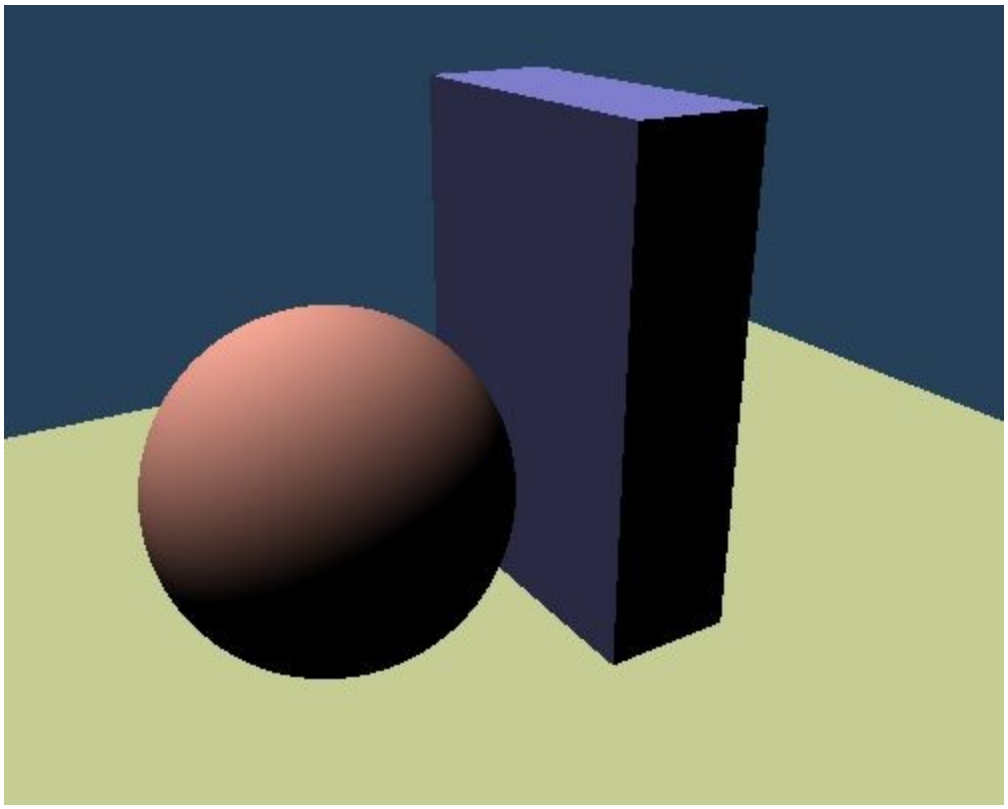
Diffuse Lighting:

While implementing diffuse lighting seemed (and theoretically is) straight forward, I ran into a few problems while implementing it.

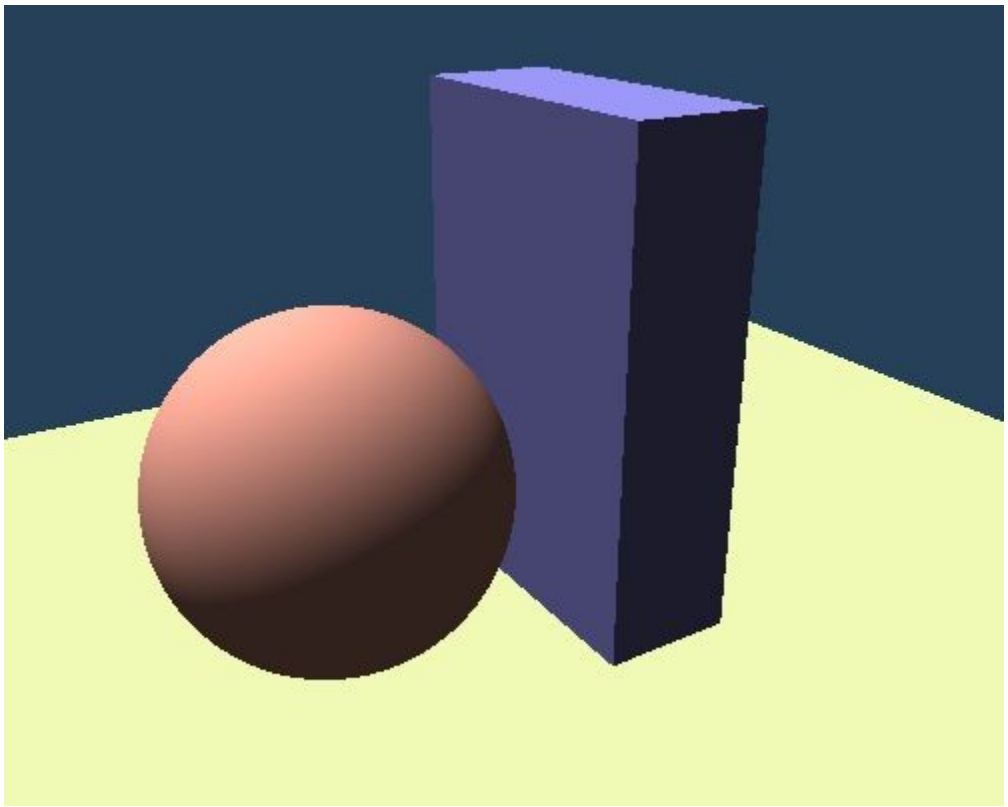
The first problem was that a bug had existed from my ray/box intersection code which didn't correctly set the normals at intersection points. Trouble shooting this required creating the image with normals for colors as to observe how they were set, and what tests to set them correctly were working, and which weren't. In the end, the problem was due to having a subtraction in a few tests where I needed an addition, but none the less the trouble shooting was good to learn and lent to the creation of the normal-mapped image below:



After fixing the normal bug, the next "difficulty" came from simply misunderstanding the use of certain material variables. While my understanding of the physical phenomena was correct, I was assuming certain variables represented something other than what they were. In this example, I first assumed the diffuse property was a percentage of diffuse light that would be "reflected" or sent out. However, this wasn't the case, and when including it in my calculations, it resulted in a darker scene. Finally upon removing that value from my calculations I was able to obtain the correctly diffused and lit scene. Both scenes are shown below.



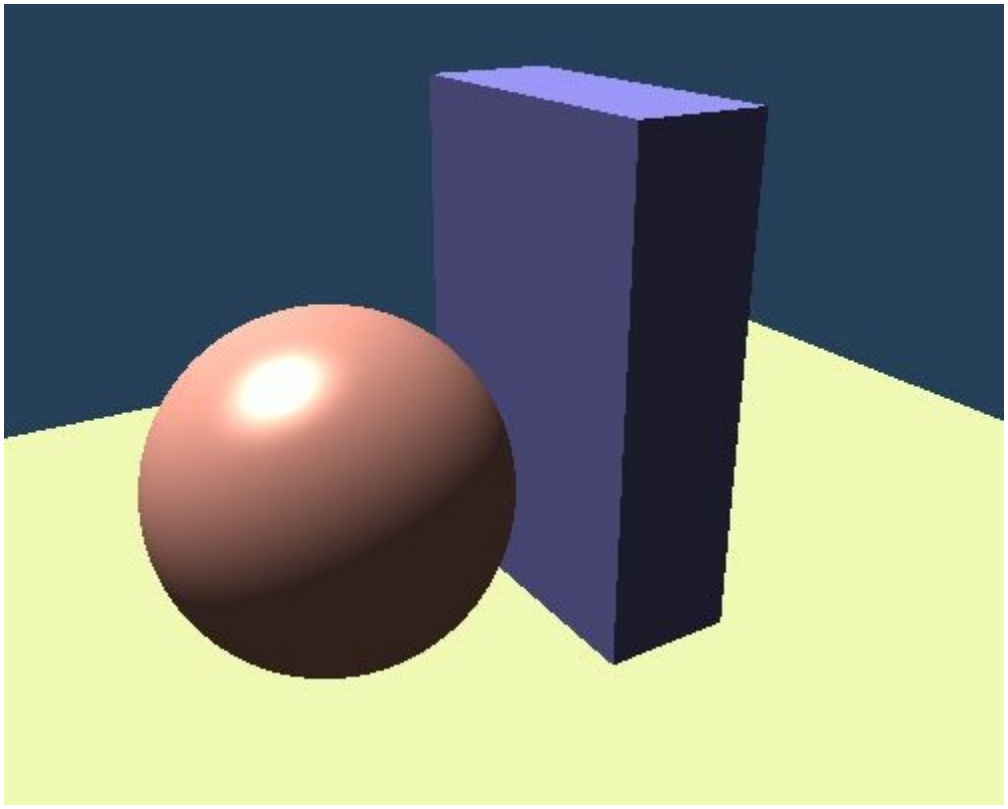
Incorrectly diffused scene. Notice it's too dark.



Correctly diffused scene.

Specular Lighting:

Implementing Specular lighting was something that helped find the normal problem with ray/box intersections and certain areas were incorrectly lit, as well as helped define a few extra tests for specular lighting such as if a phongExp is 0, don't apply specular lighting (rather than applying a sharp [white] specular which happens with low phong exponents). However, specular lighting may have been one of the easiest parts to implement simply because I correctly understood which variables to use where and thus the related troubleshooting was simple. Below is the scene correctly rendered with specular lighting up until this point.



Ambient light, diffuse light and specular light.

Shadows:

To implement shadows, a test was to be applied before adding diffuse and specular lighting. This test shot a ray ("shadow ray") from the light position (for each light), and a test was done on the intersection point return in hitInfo. If the ray shot from the light hit the object that the ray shot from the view position hit, as well as hit it in the same point, then the point was considered to not be shadowed, and thus diffuse light and specular light are added. If the shadow ray hit a different object, or the same object at a different point as the view ray, then the point is considered to be under shadow and only the ambient color/light is considered for this point.

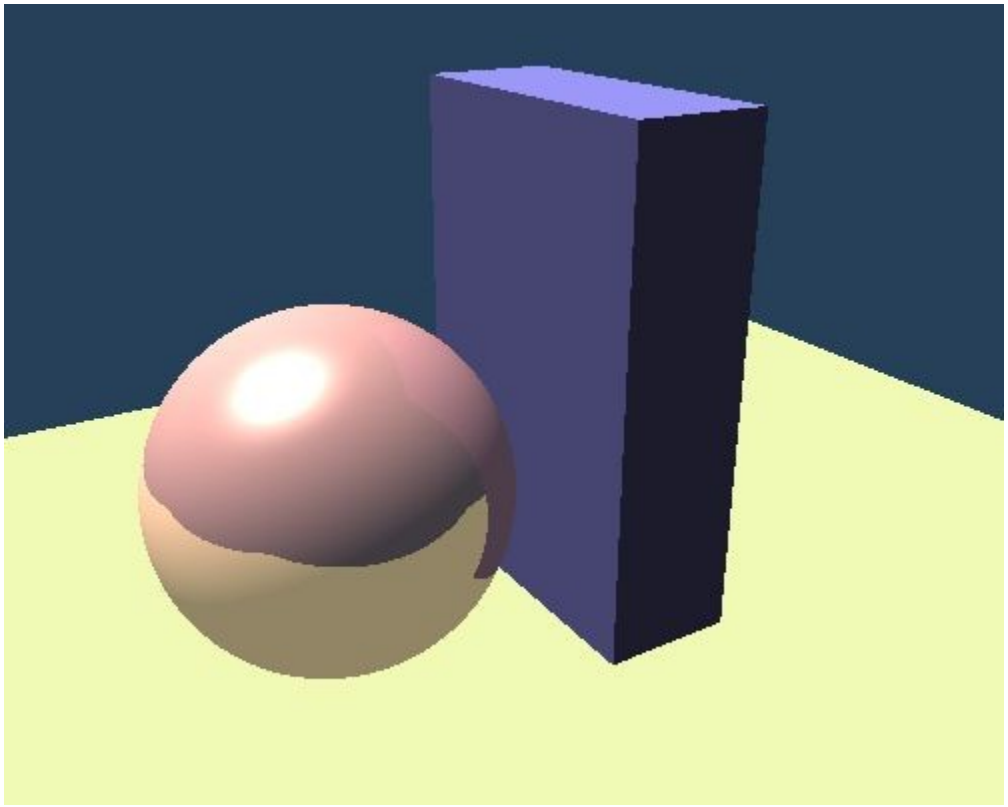
Unfortunately due to time constraints and technical difficulties, this section didn't get finished. The idea was programmed, but for some undetermined reason so far, the

casting of the ray from the light source never returns an intersection with any object. Reason still to be determined.

Another approach was also tested but didn't get debugged or end up working. The second approach is to shoot a ray from the original intersection point towards the light. If this ray intersects anything, then a test was performed on the distance of that ray to the intersected object. If that object is farther than the light source, then it doesn't cast a shadow, otherwise it does. If there is no intersection, then there is no object on that path and there is no shadow.

Reflection:

Idea behind reflections is simple. Given incident ray, compute reflected ray, trace that ray and multiply the result by material's reflectivity property. Luckily, in implementation this was quite simple. The only "difficulty" was that if you start at the point on the object that is gaining the reflection, you may discover spots if you don't move a short distance on the reflected ray first. This is due to the point being inside the object and thus reflecting off itself internally. Reflections were actually the easiest and most straight forward part to code.



Reflection implementation without shadows.

Supplementary Information:

Code was added to multiple parts of the original "toytracer." Functions which were changed are included below in full. These functions include the Shade function for the basic_shader class and the Intersect function for the block class.

basic_shader::Shade

```
Color basic_shader::Shade( const Scene &scene, const HitInfo &hit ) const
    {
    // ********* Keep as little or as much of the following code as you wish
*******
    Ray ray;
    HitInfo otherhit;
    static const double epsilon = 1.0E-4;
    if( Emitter( hit.object ) ) return hit.object->material->emission;

    Material *mat   = hit.object->material;
    Color  diffuse  = mat->diffuse;
    Color  specular = mat->specular;
    Color  color    = mat->ambient * diffuse;
    Vec3   O = hit.ray.origin;
    Vec3   P = hit.point;
    Vec3   N = hit.normal;
    Vec3   E = Unit( O - P );//O - P
    Vec3   R = Unit( ( 2.0 * ( E * N ) ) * N - E );
    Color  r = mat->reflectivity;
    double e = mat->Phong_exp;
    double k = mat->ref_index;
       Color dif = Color(0, 0, 0);
       Color spec = Color(0, 0, 0);
       Color reflect = Color(0, 0, 0);
       Color amb = mat->ambient;
       Color unscaled = Color(0, 0, 0);
       Color scaled = Color();

    if( E * N < 0.0 ) N = -N;  // Flip the normal if necessary.

    //******** FILL IN AS NEEDED ***************
    //******** FILL IN AS NEEDED ***************
    //******** FILL IN AS NEEDED ***************

    for( unsigned i = 0; i < scene.NumLights(); i++ )
    {
        const Object *light = scene.GetLight(i);
        Color emission = light->material->emission;
        AABB box = GetBox( *light );
        Vec3 LightPos( Center( box ) );


        //******** FILL IN AS NEEDED ***************
        //******** FILL IN AS NEEDED ***************
        //******** FILL IN AS NEEDED ***************


              //Cast ray from P in direction of light
              //If distance to intersection < distance to light, shadowed.
              //Otherwise there is nothing in the way between P and light
              Ray shadowRay = Ray();//new ray
```

```
            shadowRay.direction = Unit ( LightPos - P );//end - beginning; goes
towards LightPos
            shadowRay.origin = P + ( shadowRay.direction * 10E-5 );//starts at P
            shadowRay.type = shadow_ray;
            shadowRay.from = hit.object;
            HitInfo shadowHitInfo;
            shadowHitInfo.ignore = shadowRay.from;
            bool shadowed = true;
            shadowed = scene.Cast(shadowRay, shadowHitInfo);//returns true if
theres an intersection

            //if cast hit something, check that what it hit (distance) is closer
than light source.
            //if it hit something not closer than the light, set to false.

            if( dist( (shadowHitInfo.distance * Unit( LightPos - P ) + P),
LightPos) < 10E-4 && dist( (shadowHitInfo.distance * Unit( LightPos - P ) + P),
LightPos) > -10E-4 )
            {
                  shadowed = false;
            }
            if( shadowed == true )//if shadowed, don't modify dif, spec or ref
            {

            }
            else
            {

                  //deal with summations for diffuse and specular
                  //as they both address L: the direction/emition of
                  //light sources
                  //Diffuse
                  double dot = ( N * Unit ( LightPos )   );
                  if( dot >= 0 )
                  {
                        dif = dif + dot * ( diffuse );
                  }

                  //Specular
                  double phongDot = ( Unit( LightPos ) * R );
                  if(phongDot >= 0 && ( mat->Phong_exp > 10E-10 ) )
                        spec = spec + ( emission * (mat->specular) ) *
                        ( pow( phongDot, mat->Phong_exp ) );


            }//end shadowing else


      }

   //******** FILL IN AS NEEDED ***************
   //******** FILL IN AS NEEDED ***************
   //******** FILL IN AS NEEDED ***************

      //deal with reflected light (recursive rays)
      //No reason to trace if reflection co-efficient is 0 (or close enough to
      it)
      if( r.blue > 10E-8 && r.green > 10E-8 && r.red > 10E-8 )
```

```
    {
        Ray reflectRay;
        reflectRay.origin = P + ( R * 10E-4 );
        reflectRay.direction = R;
        reflectRay.generation = hit.ray.generation + 1;
        reflectRay.from = hit.object;
        reflect = scene.Trace(reflectRay);
        reflect = r * reflect;
    }


    //******** FILL IN AS NEEDED ***************

    //normal mapping for testing (Need to fix normals for ray/box intersection
    //  currently incorrect normals are assigned - FIXED)
    //if(N.x < 0)
    //     N.x = N.x * -1;
    //unscaled.red = N.x;
    //if( N.y < 0 )
    //     N.y = N.y * -1;
    //unscaled.green = N.y;
    //if(N.z < 0)
    //     N.z = N.z * -1;
    //unscaled.blue = N.z;

    //sum ambient, diffuse, specular and reflected
    //and return that as result color

    unscaled = (color + dif + spec + reflect);//Reflection Finished

    double max;
    double scale;
    max = unscaled.blue;
    if(unscaled.green > max)
        max = unscaled.green;
    if(unscaled.red > max)
        max = unscaled.red;
    if(max > 255.0)
    {
        scale = 255.0 / max;
        scaled.blue = unscaled.blue * scale;
        scaled.green = unscaled.green * scale;
        scaled.red = unscaled.red * scale;
    }
    else
    {
        scaled = unscaled;
    }

    return scaled;
}
```

Block::Intersect

```
bool Block::Intersect( const Ray &ray, HitInfo &hitinfo ) const
    {
       //
    //******** FILL IN AS NEEDED ***************
    //
       double tMax = DBL_MAX;//distance along ray
       double tMin = DBL_MIN;//distance along ray
       const Vec3 Ac( Max.x - ((Max.x-Min.x)/2), Max.y - ((Max.y-Min.y)/2), Max.z
- ((Max.z-Min.z)/2) );//center of box
       const Vec3 O( ray.origin );//ray origin
       const Vec3 D( ray.direction );//ray direction
       Vec3 P = Ac - O;//point
       double halfs[3];
       halfs[0] = (Max.x-Min.x)/2;
       halfs[1] = (Max.y-Min.y)/2;
       halfs[2] = (Max.z-Min.z)/2;
       Vec3 a[3];
       a[0] = Vec3(1, 0, 0);
       a[1] = Vec3(0, 1, 0);
       a[2] = Vec3(0, 0, 1);
       a[0] = Unit( a[0] );
       a[1] = Unit( a[1] );
       a[2] = Unit( a[2] );
       double e, f, t1, t2, temp;
       bool updated = false;
       int minNormIndex = 0;
       int maxNormIndex = 0;

       for(int i = 0; i < 3; i++)
       {
              e = a[i] * P;
              f = a[i] * D;
              if(abs(f) > DBL_EPSILON)
              {
                     t1 = (e + halfs[i]) / f;
                     t2 = (e - halfs[i]) / f;
                     if(t1 > t2)
                     {
                            temp = t1;
                            t1 = t2;
                            t2 = temp;
                     }
                     if(t1 > tMin)
                     {
                            tMin = t1;
                            minNormIndex = i;
                     }
                     if(t2 < tMax)
                     {
                            tMax = t2;
                            maxNormIndex = i;
                     }
                     if(tMin > tMax)
                     {
                            return false;
```

```
                        }
                        if(tMax < 0)
                        {
                                return false;
                        }
                }
                else if( (-1)*e - halfs[i] > 0 || (-1)*e + halfs[i] < 0 )
                {
                                return false;
                }
        }
        if(tMin > 0)
        {
                if(tMin < hitinfo.distance)
                {
                        //return true w/ tMin
                        hitinfo.distance = tMin;
                        hitinfo.point = ray.origin + (ray.direction * tMin);

                        if( hitinfo.point.x < (Max.x + 10E-10) && hitinfo.point.x >
(Max.x - 10E-10) )
                        {
                                hitinfo.normal = Unit( Vec3(1, 0, 0) );
                        }
                        else if( hitinfo.point.x < (Min.x + 10E-10) && hitinfo.point.x
> (Min.x - 10E-10) )
                        {
                                hitinfo.normal = Unit( Vec3(-1, 0, 0) );
                        }
                        else if( hitinfo.point.y < (Max.y + 10E-10) && hitinfo.point.y
> (Max.y - 10E-10) )
                        {
                                hitinfo.normal = Unit( Vec3(0, 1, 0) );
                        }
                        else if( hitinfo.point.y < (Min.y + 10E-10) && hitinfo.point.y
> (Min.y - 10E-10) )
                        {
                                hitinfo.normal = Unit( Vec3(0, -1, 0) );
                        }
                        else if( hitinfo.point.z < (Max.z + 10E-10) && hitinfo.point.z
> (Max.z - 10E-10) )
                        {
                                hitinfo.normal = Unit( Vec3(0, 0, 1) );
                        }
                        else if( hitinfo.point.z < (Min.z + 10E-10) && hitinfo.point.z
> (Min.z - 10E-10) )
                        {
                                hitinfo.normal = Unit( Vec3(0, 0, -1) );
                        }
                        else
                        {
                                hitinfo.normal = Unit( Vec3(1, 0, 0) );
                        }

                        hitinfo.object = this;
                        updated = true;
                }
        }
```

```
	else
	{
		if(tMax < hitinfo.distance)
		{
			//return true w/ tMax
			hitinfo.distance = tMax;
			hitinfo.point = ray.origin + ray.direction * tMax;

			if( hitinfo.point.x < (Max.x + 10E-10) && hitinfo.point.x >
(Max.x - 10E-10) )
			{
				hitinfo.normal = Unit( Vec3(1, 0, 0) );
			}
			else if( hitinfo.point.x < (Min.x + 10E-10) && hitinfo.point.x
> (Min.x - 10E-10) )
			{
				hitinfo.normal = Unit( Vec3(-1, 0, 0) );
			}
			else if( hitinfo.point.y < (Max.y + 10E-10) && hitinfo.point.y
> (Max.y - 10E-10) )
			{
				hitinfo.normal = Unit( Vec3(0, 1, 0) );
			}
			else if( hitinfo.point.y < (Min.y + 10E-10) && hitinfo.point.y
> (Min.y - 10E-10) )
			{
				hitinfo.normal = Unit( Vec3(0, -1, 0) );
			}
			else if( hitinfo.point.z < (Max.z + 10E-10) && hitinfo.point.z
> (Max.z - 10E-10) )
			{
				hitinfo.normal = Unit( Vec3(0, 0, 1) );
			}
			else if( hitinfo.point.z < (Min.z + 10E-10) && hitinfo.point.z
> (Min.z - 10E-10) )
			{
				hitinfo.normal = Unit( Vec3(0, 0, -1) );
			}
			else
			{
				hitinfo.normal = Unit( Vec3(1, 0, 0) );
			}

			hitinfo.object = this;
			updated = true;
		}
	}


    return updated;
    }
```