Calculate Reflection and Refraction

```
void lambertianMaterial::ReflectedColor(rgb &outA, rgb &outD, rgb &outS, const
	lightOutput &lightVal, const intersection &inter){
	outA = rgb::black;
	outD = inter.diff;
	outS = rgb::black;

	// Assume that these are all outgoing from the surface and are already normalized
	vector L = lightVal.L;
	vector N = inter.n;
	vector V = inter.v;

	// Get the forward-facing normal
	faceforward(N,V);

	// calculate ambient reflection from ambient light source and diffuse reflectance
	outA = lightVal.a * inter.diff;
	// calculate the dot product between L and N
	double LdotN = Dot(L,N);
	// calculate diffuse reflection
	outD = LdotN * (inter.diff * lightVal.d);
}

void phongMaterial::ReflectedColor(rgb &outA, rgb &outD, rgb &outS, const lightOutput
	&lightVal, const intersection &inter){
	rgb diffuse, specular;
	shape *shp = inter.s;

	outA = rgb::black;
	outD = rgb::black;
	outS = rgb::black;

	// Assume that these are all outgoing from the surface and are already normalized
	vector L = lightVal.L;
	vector N = inter.n;
	vector V = inter.v;

	// Get the forward-facing normal
	faceforward(N,V);

	// calculate ambient reflection from ambient light source and diffuse reflectance
	outA = lightVal.a * inter.diff;
	// calculate the dot product between L and N
	double LdotN = Dot(L,N);
	// calculate diffuse reflection
	outD = LdotN * (inter.diff * lightVal.d);

	// calculate the reflection R  of L about N
	vector R = 2 * L.Proj(N) - L;
	// calculate the dot product of R and V
	double RdotV = Dot(R,V);
	// calculate the specular reflection
	if(RdotV >= 0){
		outS = inter.spec * lightVal.s * pow(RdotV, inter.shiny) ;
	}

}
```

Calculate Reflection and Refraction

```
void blinnMaterial::ReflectedColor(rgb &outA, rgb &outD, rgb &outS,const lightOutput
       &lightVal, const intersection &inter){
       rgb diffuse, specular;

       outA = rgb::black;
       outD = rgb::black;
       outS = rgb::black;

       // Assume that these are all outgoing from the surface and are already normalized
       vector L = lightVal.L;
       vector N = inter.n;
       vector V = inter.v;

       // Get the forward-facing normal
       faceforward(N,V);
       // calculate ambient reflection from ambient light source and diffuse reflectance
       outA = lightVal.a * inter.diff;
       // calculate the dot product between L and N
       double LdotN;
       // calculate diffuse reflection
        outD = LdotN * (inter.diff * lightVal.d);
       // Calculate the half-angle vector and normal
       vector H = (L + V)/2;
       H.Normalize();
       // Compute specular reflection from half-angle vector
       double HdotV = Dot(H,V);
       outS = inter.spec * lightVal.s * pow(HdotV, inter.shiny) ;
}

void glass::ReflectedColor(rgb& outA, rgb &outD, rgb &outS, const lightOutput &lightVal,
       const intersection &inter){
       rgb diffuse, specular;
       shape *shp = inter.s;

       outA = rgb::black;
       outD = inter.diff;
       outS = rgb::black;

       // Assume that these are all outgoing from the surface and are already normalized
       vector L = lightVal.L;            // L is outgoing
       vector N = inter.n;               // N is outward facing
       vector V = inter.v;               // V is outgoing

       // Get the forward-facing normal
       N = faceforward(N, V);
       // calculate ambient reflection from ambient light source and diffuse reflectance
       outA = lightVal.a * inter.diff;
       // calculate the dot product between L and N
       double LdotN = Dot(L,N);
       // calculate diffuse reflection
       outD = LdotN * (inter.diff * lightVal.d);
       // calculate the reflection R  of L about N
       vector R = 2 * L.Proj(N) - L;
       // calculate the dot product of R and V
       double RdotV = Dot(R,V);
```

Calculate Reflection and Refraction

```
// calculate the specular reflection -- Don't highlight the dark side
if(RdotV >= 0){
	RdotV = smoothstep(.96,.99,RdotV);
	outS = inter.spec * lightVal.s * pow(RdotV, inter.shiny) ;
}
if (world != NULL && level < maxLevel){
	level++;
	rgb transmitRayResult, reflectRayResult;

	vector V, N;
	V = inter.v;
	N = inter.n;

	// compute the position of the intersection
	// reflect the reflection of v about the surface normal
	ray reflectRay;
	reflectRay.v = reflect(inter.n,inter.v);
	reflectRay.p = inter.p + ALMOST_ZERO * reflectRay.v;

	// coefI:  Index of refraction of material that the ray is leaving
	// coefT:  Index of refraction of material that transmitted ray will enter
	double coefI, coefT;

	//vector Nv;
	if(Dot(inter.v,N) > 0 ){
		coefI = ambientN;
		coefT = materialN;
		//Nv = -inter.n;
	}else{
		coefI = materialN;
		coefT = ambientN;
		//Nv = inter.n;
	}

	// compute normal in direction of v
	vector Nv;
	Nv = faceforward(N,inter.v);


	// compute the (acute) angle between Nv and V
	double thetaI = acos(Dot(inter.v,Nv));
	//		replace current assignment statement for sinThetaT
	//		with application of Snell's law to compute the sine of the angle
	//		of the transmitted ray
	double sinThetaT = (coefI/coefT) * sin(thetaI);

	double fkR, fkT;

	if (fabs(sinThetaT) < .99999)// Then thetaT will be < 90 degrees
	{
		// compute corresponding angle of sinThetaT
		double thetaT =  asin(sinThetaT);

		// normalize the part of V that's perpendicular to Nv
		vector tangentV = V.Perp(Nv);
		tangentV.Normalize();
```

Calculate Reflection and Refraction

```
                        //        compute the point of intersection and
                        //        direction of the transmitted ray
                        ray transmitRay;

                        transmitRay.v = -Nv * cos(thetaT) - tangentV * sin(thetaT);
                        transmitRay.p = inter.p + ALMOST_ZERO * transmitRay.v;
                        //         compute Reflective Intensity (fkR) and
                        //         Transparent Intensity (fkT) according to
                        //         Fresnel's equations

                        double fresnel = .5 * ( ( pow( tan(thetaI-thetaT) ,2) /  pow(
                        tan(thetaI+thetaT) ,2) ) + ( pow( sin(thetaI-thetaT) ,2) /  pow(
                        sin(thetaI+thetaT) ,2) ) );

                        fkR = kR * fresnel;
                        fkT = kT * (1-fresnel);

                        if (IsNotZero(fkT * kT)){
                                transmitRayResult = fkT * kT * world->TraceRay(transmitRay);
                        }else{
                                transmitRayResult = rgb::black;
                        }
                }
                else// Total internal reflection, thetaT = 90 degrees, no transmission
                {
                        fkR = sqr(coefI - coefT)/sqr(coefI + coefT);
                        transmitRayResult = rgb::black;
                }
                if (IsNotZero(fkR * kR)){
                        reflectRayResult = fkR * kR * world->TraceRay(reflectRay);
                }else{
                        reflectRayResult = 0.0;
                }

                outS += reflectRayResult + transmitRayResult * transmitColor;
                level--;
        }
}
```