Porting RSL to C++

Ryusuke Villemin, Christophe Hery

Pixar Technical Memo 12-08

1 Introduction

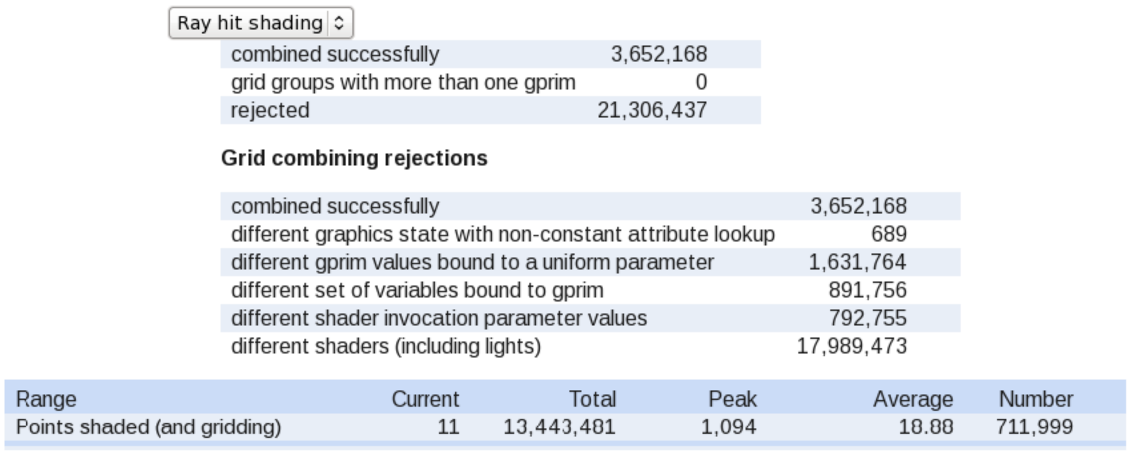
In a modern renderer, relying on recursive ray-tracing, the number of shader calls increases by one or two order of magnitude compared to a straighforward rasterizer dealing only with camera visible objects. Recognizing the potential overhead of RSL parsing in all these shader calls, this report evaluates different C++ pre-compiled alternatives.

Most of our current shaders are written using RSL. The main advantage is ease of development and use: the shader writer only needs to code a standard scalar version and the compiler is going to parallelize it to run on multiple shading points of a grid. Since RSL is byte-code compiled, it can suffer a performance penalty compared to binary compiled languages. However many shading points are run in parallel and thus this cost has always been assumed to be leveraged.

Unfortunately with the introduction of ray-tracing, it is getting more and more difficult to shade multiple points at once, since divergent rays are likely to hit completely different objects and shaders. In practice, for an indoor scene, where almost every ray hits something, we found that 80% of the shaders can't be combined.

As shown in the statistics, the average grid size can drop to a few dozen shading points (or even less). In this context the overhead of interpreting RSL becomes non negligible.

In part 2, we will discuss different versions of our shader and the cost of porting it to different languages. In part 3, we will present the results of those different versions, and then conclude in part 4.



combined successfully	3,652,168
grid groups with more than one gprim	0
rejected	21,306,437

Grid combining rejections

combined successfully	3,652,168
different graphics state with non-constant attribute lookup	689
different gprim values bound to a uniform parameter	1,631,764
different set of variables bound to gprim	891,756
different shader invocation parameter values	792,755
different shaders (including lights)	17,989,473

Range	Current	Total	Peak	Average	Number
Points shaded (and gridding)	11	13,443,481	1,094	18.88	711,999

Figure 1: Grid Statistics

2 Implementation

2.1 RSL to C++

In RSL, a shader is written as it is handling only one shading point, prman will automaticaly perform the same operations on all the shading points of the grid. The shader only have to differentiate, per grid operation and per shading point operation by using the 2 keywords `uniform` and `varying` in front of each variables.

Usually the code is divided more or less clearly in 3 components :

- operations that need to run once per grid `do_per_grid_operations()`
- operations that need to run per shading point `do_per_shading_point_operations()`
- operations needed per samples per shading point `do_per_sample_per_shading_point_operations()`

Those 3 functions have dependencies and need to be called in this particular order. In all the following pseucode, we will consider that we have an M points grid and we are taking N samples on each of these shading points.

```
uniform var_grid = do_per_grid_operations()

varying var_shading_point = do_per_shading_point_operations(var_grid)

for (i=0..N)
{
    do_per_sample_per_shading_point_operations(var_grid, var_shading_point)
}
```

When porting this code to C++, it is now up to the shader writer to perform the loop over the shading points of the grid. The 3 different functions are now in 3 different parts of the loops (the grid operations are completely outside).

```
var_grid = do_per_grid_operations()

for (j=0..M)
{
    var_shading_point = do_per_shading_point_operations(var_grid)

    for (i..N)
    {
        do_per_sample_per_shading_point_operations(var_grid, var_shading_point)
    }
}
```

The translation to C++ is pretty straighforward and easy, RSL syntax being very close to C in the first place. Just porting the code to C++ gives more than 20% speedup for large grids and more than 60% on small grids.

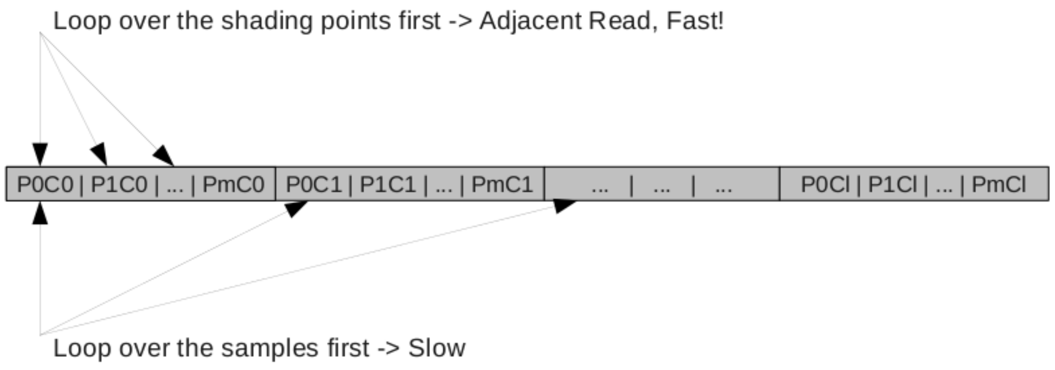

Loop over the shading points first -> Adjacent Read, Fast!
P0C0 | P1C0 | ... | PmC0
P0C1 | P1C1 | ... | PmC1
... | ... | ...
P0Cl | P1Cl | ... | PmCl
Loop over the samples first -> Slow

Figure 2: prman memory layout

2.2 Optimizing C++

The previous implementation is far from being optimal, the biggest reason is related to how the arrays are handled by prman. For a varying array C of size l, the memory layout inside the renderer is:

P0C0 P1C0 . . . PmC0 P0C1 P2C1 . . . . PmC1 P0C2 . . . . . . PmCl

That means it is far more efficient to perform adjacent read per shading points, than read per samples. When reading per samples, we have to jump m elements every access. A naive way to do per shading point reads is to just interchange the loops:

```
var_grid = do_per_grid_operations()

for (i=0..N)
{
    for (j=0..M)
    {
        var_shading_point = do_per_shading_point_operations(var_grid)
        do_per_sample_per_shading_point_operations(var_grid, var_shading_point)
    }
}
```

Unfortunately the `do_per_shading_point_operations()` is now repeated NxM times instead of the needed M times. This can lead to slower execution than our first version. We have to extract that code and make a second loop:

```
var_grid = do_per_grid_operations()

var_shading_point[M]

for (j=0..M)
{
    var_shading_point[j] = do_per_shading_point_operations(var_grid)
}

for (i=0..N)
{
    for (j=0..M)
    {
        do_per_sample_per_shading_point_operations(var_grid, var_shading_point[j])
    }
}
```

One difficulty here is that in `do_per_sample_per_shading_point_operations()`, we usually need to use the result from `do_per_shading_point_operations()`. We have to store the values computed in the first loop to be used in the second loop. Since we want to avoid doing a malloc every shader call (which can be called billions of time), we reserve a memory pool in a per thread memory storage. This is possible since the shader is not reentrant.

Changing the loop order gives us an additional 20% speedup for large grids and 10% for small grids.

2.3 C++ to ispc

ispc is the new intel spmd program compiler. It uses a C like syntax and generates a .o file that we can link as usual into a DSO. The difference between ispc and other parallel programming librairies is that ispc operates on SIMD lanes, while others are focusing on utilizing multicores. ispc performs optimizations and tranforms the program in a vector intermediate representation of LLVM. Then the final compilation is handled by LLVM.

Internally ispc is going to run multiple instances of the program at once. The group of running program instances is called a gang.

Porting C++ code to ispc is really easy, it is almost a carbon copy-paste. The only changes are the loop declarations: instead of using the classic `for()` loop, we use the ispc specific `foreach()`. Also we now have 2 extra qualifiers for variables: `uniform` and `varying`.

One thing to be aware of is that the `uniform` and `varying` qualifiers have a slightly different meaning in ispc and in RSL. In RSL varying means varying between shading points. In ispc, varying means varying between program instances within a gang. So a varying element in RSL, is likely to be an uniform array in ispc. Pointers to this uniform array are going to be varying, since every program instance is going to process different locations in this array.

To summarize :

- all the input variables including arrays are uniforms, RSL's varying inputs become uniform arrays
- all the output variables including arrays are uniforms, RSL's varying outputs become uniform arrays
- all intermediate loop dependent variables are varying like in RSL
- you should specify uniform for all loop independent variables, ispc can benefit from those by optimizing uniform code

```
uniform var_grid = do_per_grid_operations()

uniform var_shading_point[m]

foreach (j=0..m)
{
    var_shading_point[j] = do_per_shading_point_operations(var_grid)
}

foreach (i=0..n, j=0..m)
{
    do_per_sample_per_shading_point_operations(var_grid, var_shading_point[j])
}
```

Just porting the code as-is to ispc provides a 30% speedup for large grids and 20% for small grids.

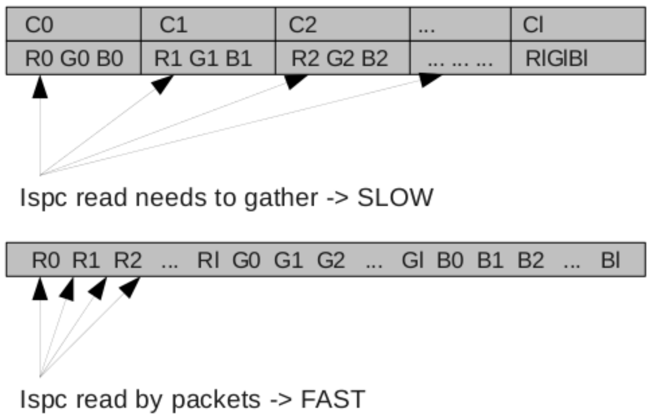

C0
C1
C2
...
Cl
R0 G0 B0
R1 G1 B1
R2 G2 B2
... ...
RlGlBl
Ispc read needs to gather -> SLOW
R0 R1 R2 ... Rl G0 G1 G2 ... Gl B0 B1 B2 ... Bl
Ispc read by packets -> FAST

Figure 3: ispc memory access

2.4 Optimizing ispc

We can make the ispc code even faster.

`uniform` conditionals are optimized in ispc but ispc also provide an improved if for varying conditionals. These are called consistent branchings (`cif`, `cwhile`) and hint the compiler that the result is likely going to be the same for all the points. This will create a special optimized code path for this branching.

We can also optimize the memory layout to have faster SIMD loads and stores by using structure of arrays instead of arrays of structure. An array of colors is usually an array of structure of the following type:

$r0g0b0\ \ r1g1b1\ \ r2g2b2\ \ \ldots\ \ rlglbl$

That means every time we load a vector, we need to perform a gather operation. If we organize the structure to be an structure of arrays, we can benefit from faster loads:

$r0r1r2\ldots rlg0g1\ldots glb0b1\ldots bl$

Changing the array memory layout is going to have an impact on the whole shader system so this time we only modified the layout of local arrays.

By doing those 2 changes (consistent branching and optimized memory layout), we were able to get an additional 25% speedup for large grids and 15% for small grids.

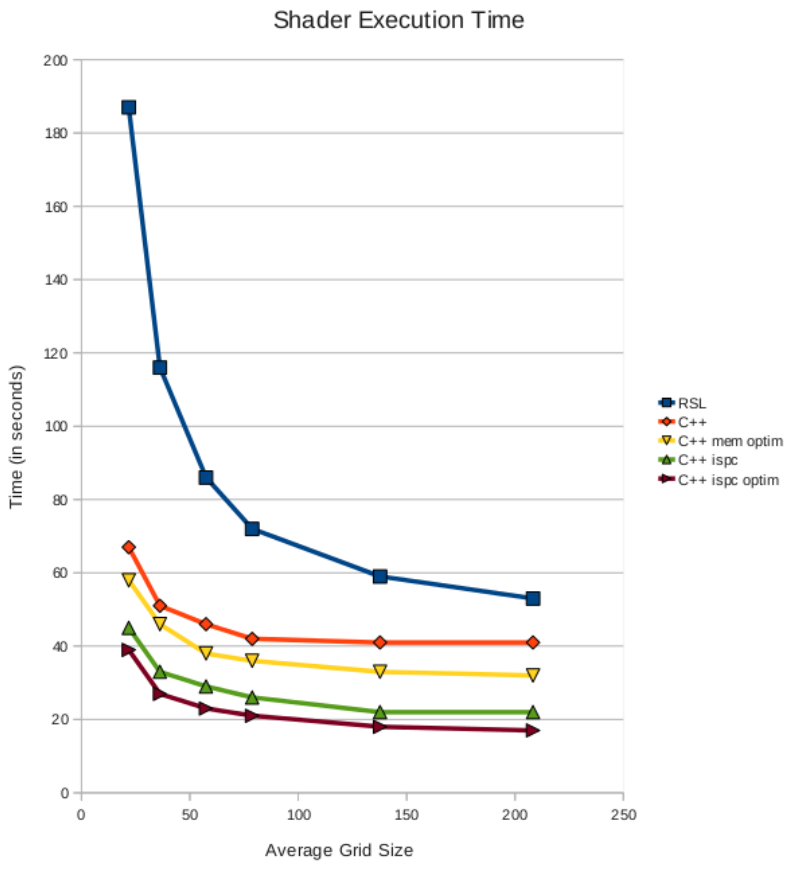


Figure 4: Shader Execution Time

3 Results

For large grids ˜200 shading points, the move from RSL to ispc gave us a 3x speedup. For small grids ˜20 shading points which are likely in a raytracing context, the speedup is more than 5x. The speedup ratios when going from C++ to the slightly optimized ispc code are respectively 2.5x and 1.7x.

Table 1: Shader Evaluation Time (in seconds).

Average Grid Size	RSL	C++	Opt. C++	ISPC	Opt. ISPC
21.88	187	67	58	45	39
36.23	116	51	46	33	27
57.49	86	46	38	29	23
78.78	72	42	36	26	21
137.74	59	41	33	22	18
208.31	53	41	32	22	17

4 Discussion

Shader execution time is largely dominant in our renders. When people are complaining about the cost of ray-tracing, it is usually the cost of ray-hit shading that is the bottleneck. In an indoor scene, the ratio ray-shading time over ray-tracing time can reach 10 pretty easily (5 hours of tracing, 50 hours of shading). Right now, this cost is masked by using exclusively the radiosity cache at ray hits, meaning we are giving up all view dependent (specular, reflection, ...) components in secondary contexts. If in the future we want to perform "full" ray-tracing, we need to address the shading time problem. Porting our shaders to ispc might be one part of the solution.

A few important remarks:

- ispc is picky in the handling of consistent ifs. Compilation time goes from 5s to 50s by using only a few of them. If replacing all the `ifs` by `cifs`, we were not even, after an hours, able to get a full compilation.
- ispc generates an optimized code for each `cif` scenario, so the .o can get pretty big. In our case the object file doubled in size, which is still reasonable, but it might become a problem if we use use too many of them.
- from the coding point of view, almost all of difficulty in porting shaders from RSL to ispc was in the second step, when reverting the 2 loops. Compared to this step, going from RSL to C++ and then from C++ to ispc is relatively easy and is likely to be be less than a day of work for any experienced programmer.
- Intel MIC/Larabee2 will have hardware gather/scatter. It may then become less important to have to convert arrays of structs (such as colors) into structs of arrays (per component arrays).
- Going from RSL to C++, we naturally get bigger gains for small grids. However, once in C++, we can still optimize large grids by vectorizing and optimizing the memory accesses. Overall we are able to speedup all cases.

A Spherelight Example

The following function `emissionAndPDF()` is from the spherelight production shader.

A.1 RSL Code

```
public void emissionAndPDF( DiffGeometryStruct diffGeom;
                            BSDFSamplesStruct bs;
                            output LightEmissionStruct le )
{
    // Initialize light emission structure
    float numValidSamples = bs->numValidGeneratedSamples;
    uniform float numSamples = getNumSamples( bs );
    le->initialize( numSamples );

    // Check whether we are inside or not
    vector lightCenterDir = m_center - diffGeom->P;
    float d2 = lightCenterDir . lightCenterDir;
    if( ( d2 - m_radius2 ) < 1e-4 )
        return;

    // Precompute extra dist falloff
    float actionfalloff = 1.0;
    if( minDist >= radius && minDist < maxDist )
        actionfalloff -= smoothstep( minDist, maxDist, sqrt(d2) );

    // Intersect light
    uniform float i;
    for( i = 0; i < numValidSamples; i += 1 )
    {
        // Direction towards the light
        vector lightDir = bs->dir[i];

        point  lightPt;
        float  pdf;
        float  dist;

        // Find point on the light
        if( IntersectAndComputePDF( diffGeom->P, lightDir, lightPt, pdf, dist ) )
        {
            le->P[i]   = lightPt;
            le->pdf[i] = pdf;
            le->Cl[i]  = m_lightColor * actionfalloff;

            if( focusFalloff > 0.0 )
            {
                // construct NdotD for focus falloff
                float NdotD = normalize( m_center - lightPt ) . lightDir;
                le->Cl[i] *= pow( NdotD, focusFalloff ) *
                               ( 1.0 + (PIDIV2-1.0) * focusFalloff );
            }
        }
    }
}
```

A.2 C++ Code

```
int EmissionAndPDF(RslContext* rslContext, int argc, const RslArg* argv[])
{
    // diffgeom point
    RslStruct diffGeom(argv[1]);
    const RslArg* diffGeomPArg(diffGeom[0]);
    float* diffGeomP;
    int diffGeomPStride;
    diffGeomPArg->GetData(&diffGeomP, &diffGeomPStride);

    // get arguments
    // ..........

    unsigned int numVals;
    const RslRunFlag *runFlags = rslContext->GetRunFlags(&numVals);

    EmissionAndPDFdata* data = getEmissionAndPDFdata(rslContext);
    data->resize(numVals);
    float *lightCenterDir = data->lightCenterDir;
    float *actionfalloff = data->actionfalloff;
    int *active = data->active;

    // non array dependent computation
    for (int g=0; g<numVals; ++g)
    {
        if (runFlags[g])
        {
            lightCenterDir[3*g]   = sphereCenter[0] - diffGeomP[3*g];
            lightCenterDir[3*g+1] = sphereCenter[1] - diffGeomP[3*g+1];
            lightCenterDir[3*g+2] = sphereCenter[2] - diffGeomP[3*g+2];

            active[g] = 0;

            float d2 = ( lightCenterDir[3*g]   * lightCenterDir[3*g]
                       + lightCenterDir[3*g+1] * lightCenterDir[3*g+1]
                       + lightCenterDir[3*g+2] * lightCenterDir[3*g+2] );

            if( ( d2 - *sphereRadius2 ) >= 1e-4 )
            {
                if( (*maxdist) <= (*radius) || ( d2 - (*maxdist2) ) <= 0.f )
                {
                    active[g] = 1;

                    // Precompute extra dist falloff
                    actionfalloff[g] = 1.f;
                    if( (*mindist) >= (*radius) && (*mindist) < (*maxdist) )
                        actionfalloff[g] -= smoothstep( (*mindist),
                                                        (*radius), sqrtf(d2) );
                }
            }
        }
    }
    // array dependent computation
    for (int i=0; i<ilsize; ++i)
    {
        int off = i*numVals;

        for (int g=0; g<numVals; ++g)
        {
            if (runFlags[g])
            {
                lpdf[g+off] = 0;

                if (active[g])
                {
                    float pdf; float dist; float lightPt[3];

                    if ( IntersectAndComputePDF(sphereCenter, *sphereRadius2,
                                                diffGeomP+3*g, bsDir+3*(g+off),
                                                lP+3*(g+off), lpdf[g+off], dist) )
                    {
                        lC[3*(g+off)]   = sphereColor[0] * actionfalloff[g];
                        lC[3*(g+off)+1] = sphereColor[1] * actionfalloff[g];
                        lC[3*(g+off)+2] = sphereColor[2] * actionfalloff[g];

                        if( (*focusfalloff) > 0.0f )
                        {
                            // construct NdotD for focus falloff
                            float radiusDir[3];
                            radiusDir[0] = sphereCenter[0] - lP[3*(g+off)];
                            radiusDir[1] = sphereCenter[1] - lP[3*(g+off)+1];
                            radiusDir[2] = sphereCenter[2] - lP[3*(g+off)+1];

                            float radiusLen = sqrt(   radiusDir[0] * radiusDir[0]
                                                    + radiusDir[1] * radiusDir[1]
                                                    + radiusDir[2] * radiusDir[2] );

                            float NdotD = (   radiusDir[0] * lightCenterDir[3*g]
                                            + radiusDir[1] * lightCenterDir[3*g+1]
                                            + radiusDir[2] * lightCenterDir[3*g+2] )
                                            / radiusLen;

                            float mixedColor[3];
                            mixedColor = powf( NdotD, *focusfalloff )
                                            * ( 1.f + (0.5f*M_PI-1.f) * (*focusfalloff) )

                            lC[3*(g+off)]   *= mixedColor[0];
                            lC[3*(g+off)+1] *= mixedColor[1];
                            lC[3*(g+off)+2] *= mixedColor[2];
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

A.3 ispc Code

```
export void ISPCSphereEmissionAndPDF(   uniform int numVals,
                                        const uniform unsigned int runFlags[],
                                        uniform Triple center[],
                                        uniform Triple diffGeomP[],
                                        uniform float d2[],
                                        uniform int active[],
                                        uniform float maxdist,
                                        uniform float maxdist2,
                                        uniform float neardist2,
                                        uniform float radius,
                                        uniform float radius2,
                                        uniform float lvalidNumSamples[],
                                        uniform int numSamples,
                                        uniform float mindist,
                                        uniform float actionfalloff[],
                                        uniform float one_over_d[],
                                        uniform float sincosThetaMax[],
                                        uniform float pdf[],
                                        ... )
{
   uniform float focusfalloffActive = ( (focusfalloff) > 0.0f  );

   // non array dependent computation
   foreach (g=0 ... numVals)
   {
      cif (runFlags[g])
      {
         lightCenterDirX[g] = sphereCenter[0] - diffGeomP[g].x;
         lightCenterDirY[g] = sphereCenter[1] - diffGeomP[g].y;
         lightCenterDirZ[g] = sphereCenter[2] - diffGeomP[g].z;

         active[g] = 0;

         float d2 = (  lightCenterDirX[g] * lightCenterDirX[g]
                     + lightCenterDirY[g] * lightCenterDirY[g]
                     + lightCenterDirZ[g] * lightCenterDirZ[g] );

         cif( ( d2 - sphereRadius2 ) >= 1e-4 )
         {
            cif( (maxdist) <= (radius) || ( d2 - (maxdist2) ) <= 0.f )
            {
               active[g] = 1;

               // Precompute extra dist falloff
               actionfalloff[g] = 1.f;
               if( (mindist) >= (radius) && (mindist) < (maxdist) )
                  actionfalloff[g] -= smoothstep( (mindist),
                                                  (radius), sqrt(d2) );
            }
         }
      }
   }
```

```
    // array dependent computation
    foreach (i=0 ... ilsize, g=0 ... numVals)
    {
        cif (runFlags[g])
        {
            int ig = g+i*numVals;

            lpdf[ig] = 0;

            cif (active[g])
            {
                float pdf; float dist; Triple lightPt;

                if ( IntersectAndComputePDF(sphereCenter, sphereRadius2,
                                            diffGeomP[g], bsDir[ig],
                                            lightPt, pdf, dist) )
                {
                    lC[ig].r = sphereColor[0] * actionfalloff[g];
                    lC[ig].g = sphereColor[1] * actionfalloff[g];
                    lC[ig].b = sphereColor[2] * actionfalloff[g];
                    lpdf[ig] = pdf;
                    lP[ig] = lightPt;

                    if( (focusfalloff) > 0.0 )
                    {
                        // construct NdotD for focus falloff
                        float radiusDir[3];
                        radiusDir[0] = sphereCenter[0] - lP[ig].x;
                        radiusDir[1] = sphereCenter[1] - lP[ig].y;
                        radiusDir[2] = sphereCenter[2] - lP[ig].z;

                        float radiusLen = sqrt(  radiusDir[0] * radiusDir[0]
                                               + radiusDir[1] * radiusDir[1]
                                               + radiusDir[2] * radiusDir[2] );

                        float NdotD = (  radiusDir[0] * lightCenterDirX[g]
                                       + radiusDir[1] * lightCenterDirY[g]
                                       + radiusDir[2] * lightCenterDirZ[g] )
                                       / radiusLen;

                        float mixedColor[3];
                        mixedColor = pow( NdotD, focusfalloff )
                                     * ( 1.f + (0.5f*PI-1.f) * focusfalloff ) );

                        lC[ig].r *= mixedColor[0];
                        lC[ig].g *= mixedColor[1];
                        lC[ig].b *= mixedColor[2];
                    }
                }
            }
        }
    }
}
```