SPHERICAL HARMONIC BASED LIGHTING

MASTER'S THESIS

RAVI ACHARYA

NCCA BOURNEMOUTH UNIVERSITY

AUGUST 2011

CONTENTS

LIST OF FIGURES

1.0 Introduction

This thesis illustrates a method that may be used for the lighting of an object through the use of spherical harmonics.

The aim of this thesis is to render an object using spherical harmonics based lighting techniques. This requires an understanding of the theory and the mathematics behind the spherical harmonics and how they are described and used.

The first section illustrates previous works into rendering objects using spherical harmonics. The reasons why each system was created is ascertained, and the advantages to such a system are speculated.

The second section contains the technical background for this project, describing the maths and the theory behind the spherical harmonics used, as well as describing their derivation from their source, the Legendre Polynomials.

The third section describes the design of the system created, the details of implementation, along with the issues encountered and their resolution.

The fourth section contains the results of this project, along with views of different lighting sources and scenes.

The fifth section concludes the project, discussing future work and the current objectives of the project.

The appendix contains guidelines on how to operate the program.

2.0 Previous work

The initial motivation for this project was an interest in the fast indirect global illumination that can be produced in Pixar Renderman Pro Server using its spherical harmonic support. This technique was used by Weta Digital in production for *Avatar* (2009, James Cameron, 20th Century Fox), in order to gather lighting directional information, and calculate occlusion "efficiently and quickly" [SAINDON] on a large number of objects, most notably the large forests rendered. Upon further research into the method used, it became clear that as opposed to using a library to recreate the technique, more would be gained from understanding the technique itself and how it can be used to greatest effect.

Spherical harmonics are very efficient data structures, and the greatest efficiency is required in video games, where cycles are at a premium, yet image quality still remains sought after. As such, the focus of the project moved to the exploration of the uses of spherical harmonics in games such as Killzone 2 and Halo 3.

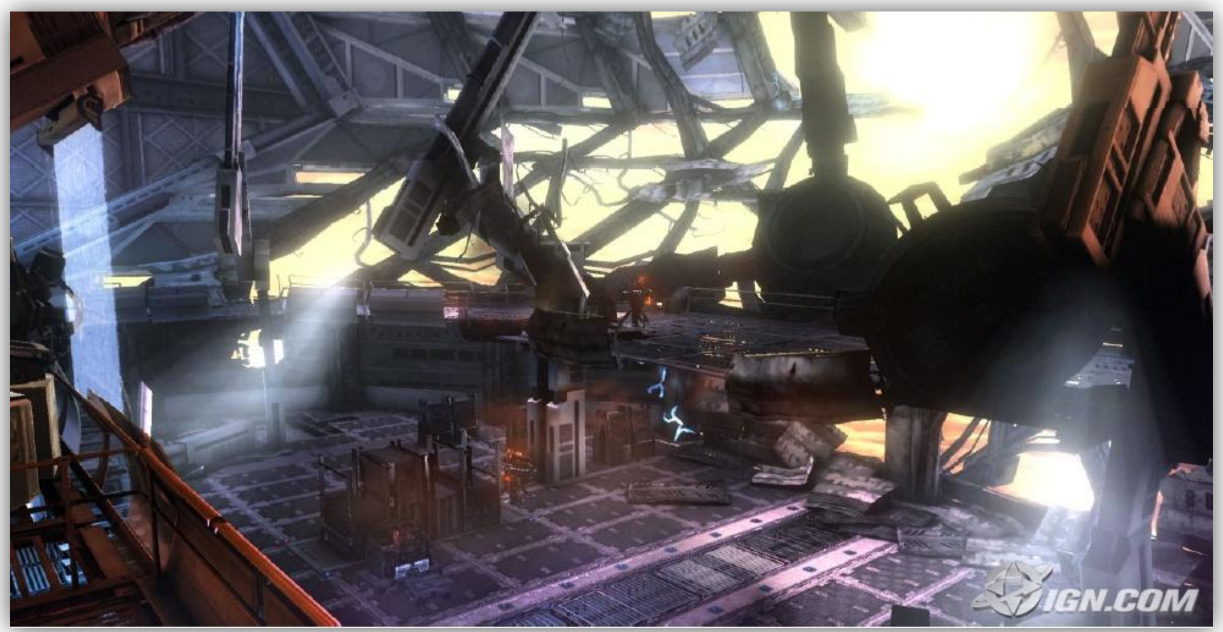

Figure 2.1: The lighting solution in Killzone 2 incorporates second order spherical harmonics allowing for fast ambient lighting. [IGNKZ2]

Spherical harmonics were used in Killzone 2 to calculate directional ambient lighting by storing this information in second order harmonics [GUERILLA07]. These light probes were placed in the levels by artists, and then presampled and stored. This is a Precomputed Radiance Transfer method (PRT) which allowed for fast harmonic sample return, allowing fast generation of diffuse lightsources. This was introduced by Peter-Pike Sloan et al, as a system for real-time global illumination [SLOAN02].

Third order spherical harmonics were again utilised in Halo 3 for diffuse lighting, with lightmaps encoded as irradiance volumes, which were calculated per "cell" and then interpolated between [CHEN08]. This resulted in an efficient dynamic diffuse lighting solution. This results in a somewhat dynamic SH lighting solution, visible in figure 2.2. Spherical harmonics remain immovable, and are generally used for static objects and environments.

Spherical harmonics have also been implemented in the Unreal engine, as a PRT method for a global illumination solver. Specifically, precomputed spherical harmonic sample volumes are used to calculate ambient occlusion, environment lighting, interreflections, and indirect lighting for dynamic objects. This was used for both occlusion and environment lighting in Mirror's Edge, pre-baked using Autodesk's "Beast" lighting engine [WFIRE11].

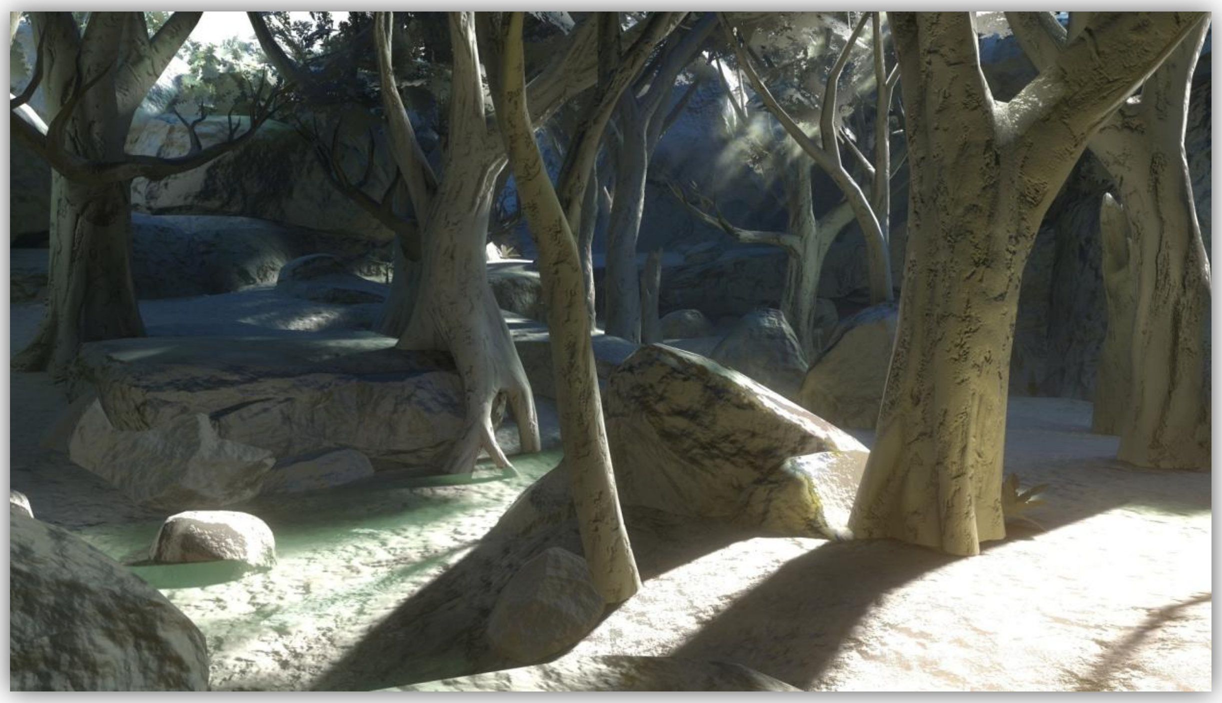

Figure 2.2: The directional diffuse lighting in Halo 3 is sampled in third order harmonics. Image courtesy of Hao Chen. [CHEN08]

The upcoming Battlefield 3 is to include dynamic radiosity by prebaking spherical harmonics samples via a series of irradiance volumes similar to Halo 3 [EINARSS10]. A system called 'Enlighten' is being used in Dice's Frostbite engine in order to light dynamic environments by allowing for groups of objects to be processed independently, causing the radiosity to be a local problem as opposed to a global operation.

The motivation for this project is to understand spherical harmonics as a system and to then create an implementation that allows the lighting of an object through spherical harmonics coefficients, created in C++ and OpenGL.

3.0 Technical Background

Spherical harmonics are widely used in physics, but have recently been used in a number of ways in computer graphics, generally as a method of lighting objects. 'Spherical harmonics' is a term given to the angular portion of the solution to Pierre-Simon Laplace's equation. Spherical harmonics (SH) is a method of storing data. A spherical function can be stored in spherical harmonics for only very small memory cost, which makes them useful for representing information which might otherwise be cumbersome to handle, such as environment maps. They have been used to represent light incidence from various directions, as storing such a map in spherical harmonics requires only a few floating points.

Ramamoorthi et al [RAMAMOOR01] introduced a method to calculate the rendering of diffuse objects under distant lighting utilising spherical harmonics. It was shown that only nine coefficients are required to render illumination with an average error in the recreated output of approximately 1%. These nine parameters can be found in the first two orders of harmonics, as Lambertian bidirectional reflectance distribution function (BRDF) attenuates non-diffuse light. As the lighting information is simplified as if it were affected by a low pass filter, the amount of input data required to perform the calculation is minimised, resulting in fewer lookups to a smaller data set. This results in a speed that allows for interactive lighting presented in a paper by Sloan [SLOAN02].

Spherical harmonics function as a data structure. The input light map that is used is encoded into a data structure. This is a process similar to a Fourier transform, with the image decomposed into a spherical representation. This representation is then used to create spherical harmonic basis functions with which coefficients can be calculated. Schönefeld describes the method used to project a function and its reconstruction, shown in Figure 3.1.

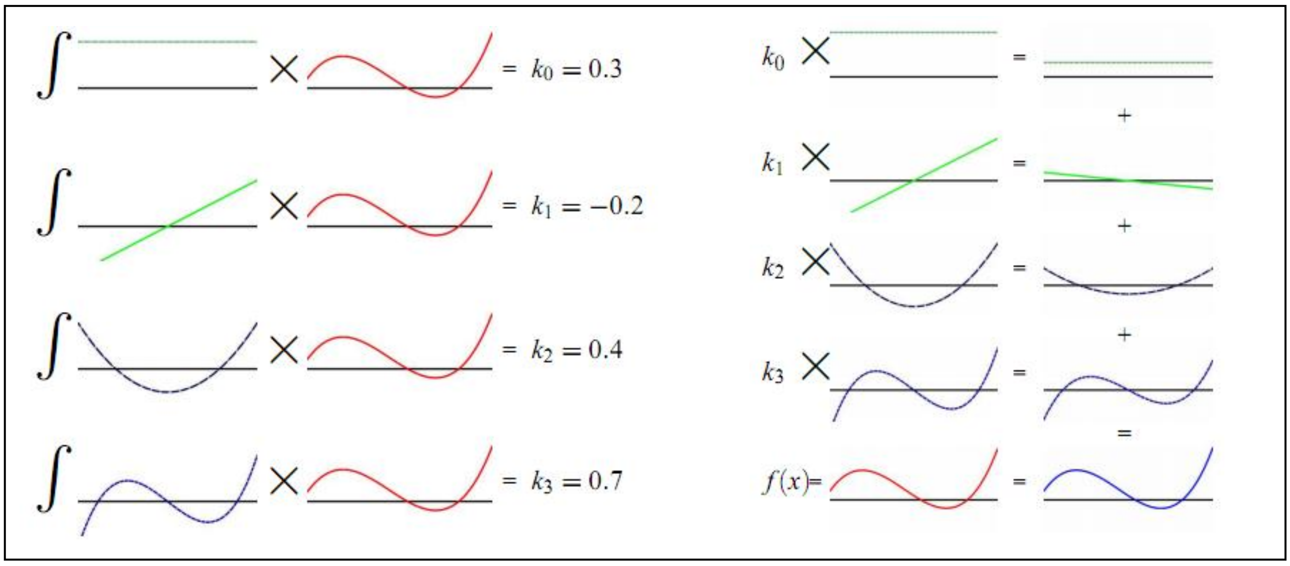

Figure 3.1: Projection of polynomial into Legendre coefficients (left) and the reconstruction of the original function by summing scaled basis functions (right). Image courtesy of Volker Schönefeld,

The reconstruction can be defined as:

$$f(x)\sum_{n=0}^{\infty} k_n p_n \tag{1}$$

where f is the given function, k_n is the Legendre coefficient and p_n is the basis function. In order to calculate the coefficients, a number of samples (based off the input map) are required which will form the basis for the spherical harmonic. To return a reconstructed function, the basis functions are individually scaled, and then summed, which results in a relatively accurate approximation of the input function.

The spherical harmonic basis functions are derived from Legendre polynomials. These Legendre polynomials are orthonormal which gives them some useful properties. These basis functions are based on *associated Legendre polynomials*, represented by $P_l^m(x)$. The function requires two arguments, l and m. In this case, l is the index of the band, and m is a positive integer between 0 and l. When $m = 0$, the associated Legendre polynomials degenerate into unassociated polynomials which once again return complex numbers, and thus cannot be computed easily. Within a band m, the polynomials are orthogonal with respect to a constant k. Between bands they are still orthogonal but with respect to a different constant k. The actual solution to the Legendre polynomials requires some complex mathematics which is not easy to compute [GREEN03,p10]. As a result, recursive definitions are used, called *recurrence relations*, which define a later polynomial based on a solution to a prior polynomial in the series.

There are three relevant *recurrence relations*:

$$P_m^m(x) = (-1)^m (2m-1)!!\,(1-x^2)^{m/2} \tag{2}$$

$$P_{m+1}^m(x) \;=\; x(2m+1)P_m^m(x) \tag{3}$$

$$(l-m)P_l^m(x) \;=\; x(2l-1)P_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x) \tag{4}$$

which are used when the associated Legendre polynomials are to be used in a computer application, since it is possible for them to be calculated and return less errors than other implementations [SCHONEFELD05,p5]. To evaluate a given value $P_l^m(x)$, the first equation (2) is used. This results in a solution where $l = m$. This is the primary equation as no prior terms are required. The second function, (3) allows the number of the band to be increased, and it requires the solution to (2) to evaluate.

The third function (4) is the main term of recurrence which requires (2) and (3) to be calculated, and generated a higher band l from the prior terms.

The order in which these are used is as follows:

1. To evaluate a given function $P_l^m(x)$, (2) is used to generate the highest P_m^m possible.

2. If this is not true, $m < l$, and (3) is used to raise the band until the required l is met.

3. (3) is performed once, and (4) is iterated until the answer is found [GREEN03,p10], [SCHONEFELD05,p5].

Samples from the input map are required in order for the SH function to have meaning, and thus they are collected and used when projecting the basis functions. A number of samples are taken from the input map, based on the accuracy of the representation required. Robin Green commented that stratified sampling is of sufficient accuracy for basic spherical harmonic lighting [GREEN03]. By collecting a random sample from each stratum, stratified sampling ensures a low sample variance, which results in a more even representation of the input data. These samples are used to gather light intensity values, which are then stored as a set of spherical co-ordinates as the angles theta (θ) and phi (φ).

The SH basis functions are created from the Legendre polynomials. The samples created earlier are used as arguments to the function, as well as m and l. However, in order to generate the SH functions, l and m are redefined. l remains a positive integer, but m can now describe a signed integer between $-l$ and l. As such the SH basis function's arguments can be shown as:

$$y_l^m(\theta,\varphi) \text{ where } l \in \mathbf{R}^+\text{, and } -l \leq m \leq l$$

Green defines a sequence y_i, which flattens y_l^m into 1D, which allows indexing [GREEN03,p12]:

$$y_l^m(\theta,\varphi) = y_i(\theta,\varphi) \tag{5}$$

where $i = l(l+1) + m$.This creates a 1D index of i, which enables much easier storage and lookup of the SH basis functions on a computer, and can be used where l and m exist in this context. The SH basis function itself is indexed by θ, φ (the sample) and by l and m.

It can be represented as follows:

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}K_l^m cos(m\varphi)P_l^m(cos\theta), & m > 0 \\ \sqrt{2}K_l^m sin(-m\varphi)P_l^{-m}(cos\theta), & m < 0 \\ K_l^0 P_l^0(cos\theta), & m = 0 \end{cases} \tag{6}$$

where y represents the SH function and P is an associated Legendre polynomial. K is a scaling factor used to normalise the functions with respect to l, and is defined as follows:

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}} \tag{7}$$

Green shows a method for calculating the scaling factor K and comments that it is often faster to precalculate the factorials used in the scaling function, as only 33 factorials are ever required, and calculating factorials is a relatively slow process. Such a function increases efficiency.

y_i can then be multiplied by the light intensity from that particular direction (θ, φ). Green derived an equation from the Monte Carlo integrator [GREEN03,p16] which describes this light integration,

$$L_l^m = \frac{4\pi}{N}\sum_{j=1}^{N} light(x_j)y_i(x_j) \tag{8}$$

in which $\frac{4\pi}{N}$ is the probability function, $light$ is the lighting contribution from the input lightmap, and x_j is the pre-calculated array of unbiased samples indexed by (5) containing (θ, φ) co-ordinates. L_l^m can also be indexed into 1D series by (5) allowing easier computation. This technique was also used by Ramamoorthi et al [SCHOLEFELD,p15] in order to calculate the light function, which combines the spherical harmonics and the input light and returns a coefficient which can be used to render a scene.

Once the coefficients have been calculated, the image can then be reconstructed as described by Ramamoorthi et al:

$$\begin{aligned} E(n) &= c_1L_{22}(x^2-y^2) + c_3L_{20}z^2 + c_4L_{00} - c_5L_{20} \\ &+2c_1(L_{2-2}xy + L_{21}xz + L_{2-1}yz) \\ &+2c_2(L_{11}x + L_{1-1}y + L_{10}z) \end{aligned} \tag{9}$$

where $E(n)$ is the diffuse lighting term, and c_x are precalculated coefficients given by Ramamoorthi et al as $\hat{A}_l$, and L_m^l are the nine lighting coefficients from third order harmonics. The image is then composed of a sum of spherical harmonic basis functions weighted by the scaling provided by the coefficients L_l^m. As these basis functions depend on the surface normal for illumination and are defined over the entire object, it is possible to show the object from any viewpoint without having to recalculate the coefficients, making the process efficient [RAMAMOOR01,p3].

4.0 Implementation

The application developed contains the framework to allow the rendering of an object light by spherical harmonics based light map.

The application class diagram is below.

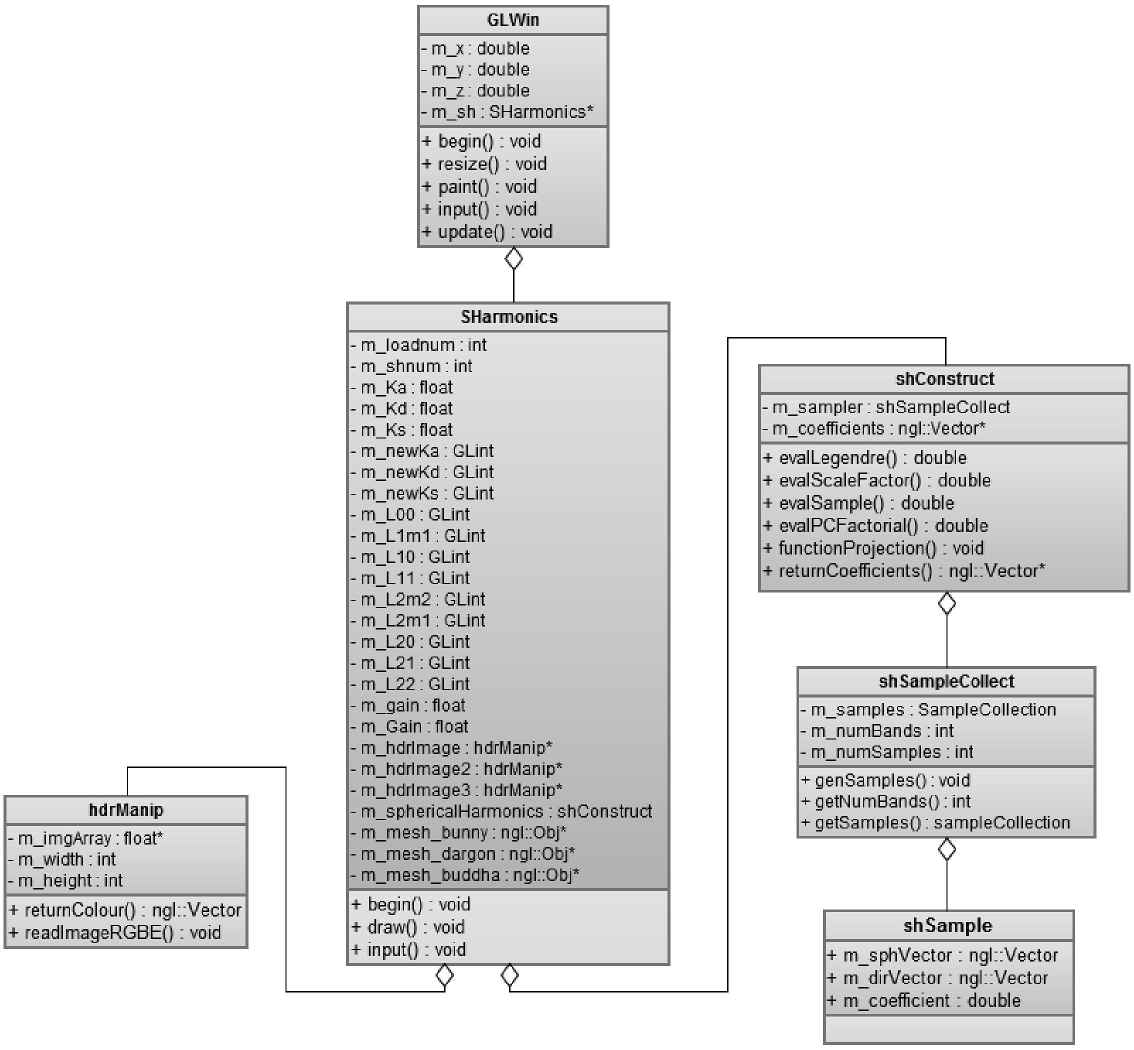

Figure 4.1 Class diagram of implemented system.

shSample exists to store the variables required by a stratified sample. A structure is declared, where a number of samples become known as a sampleCollection via a struct. shSampleCollect contains a number of samples which it creates through the method genSamples. A sample collection exists in shConstruct, which houses the spherical harmonics functions and a shSampleCollect. SHarmonics contains the coefficients and variables that are required by the shader that are calculated by shConstruct, along with methods to create them. SHarmonics also contains methods to draw and handle input. It contains a number of image files stored in hdrManips and models loaded via ngl::Obj to draw the selected item.

The application developed has a number of classes which allows the creation of spherical harmonics and the retrieval of the coefficients for use at render time.

4.1 shSample

shSample contains a direction vector 'm_dirVector', a spherical vector 'm_sphVector' and a coefficient m_coefficient pointer. The direction vector used is included from the ngl Vector library as this provides all the methods required, such as normalisation and a container for the x,y and z variables.

The spherical vector consisting of theta and phi only requires two variables, but these can safely be stored in the m_x and m_y members of ngl::Vector, with m_y simply being set to 1. This is set in shSampleCollection.cpp on line 44:

```
44      sample->m_sphVector  = ngl::Vector(theta,phi,1.0);
```

The double m_coefficient is used to store the basis function evaluation for that sample. This class is used in shSampleCollection to represent a sample.

4.2 shSampleCollect

shSampleCollect handles the creation, storage and access of the stratified samples. It contains a sampleCollection called m_samples which is a std::vector of type shSample. This allows quick lookup of samples by shConstruct which will use them. m_numBands describes the number of bands in the collection, and m_numSamples the number of samples to collect.

constructor

The constructor shCollect() takes no arguments, and only sets m_numBands to the desired number. No further initialisation is required.

genSamples(int nSamples)

This method generates a series of stratified samples as outlined in [GREEN03]. It takes the number of samples required as an integer argument. A pseudocode breakdown of the method follows:

```
{
        m_numsamples set to input number
        numreqsamples set
        1/n probability based off numreqsamples
        for 0 to numreqsamples
        {
                for 0 to numreqsamples
                        {
                                gen random x based off 1/n
                                gen random y based off 1/n
                                set theta based off x
                                set phi based off y
                                create new sample
                                set sample.sphvector variables based off theta and phi
                                convert sphvector to dirvector
                                set sample.dirvector based on sphvector
                                compute coefficients based on sample
                                add sample to sample list
                        }
        }
}
```

This results in a number of stratified samples stored in the sampleCollection m_samples.

getNumBands()

This method requires no arguments and returns the number of samples in the sample collection as an integer value. This is used in shCollect::projectFunction to reconstruct the amount of samples on which projectFunction will operate.

getSamples()

This method requires no arguments and returns m_samples as a sampleCollection. It is called from shConstruct::projectFunction to gain the sample collection in a local context, where it is used to gather light information from the input map based on the sample.

4.3 shConstruct

shConstruct contains the methods required to evaluate the SH basis functions as well as project spherical functions. It contains m_sampler, an instance of the shSampleCollect class, and uses this to create samples, evaluate their coefficients and store them. The scaled coefficients after function projection are stored in m_coefficients during projectFunction().

constructor

The constructor shConstruct() requires an integer argument of the number of samples required. This then calls genSamples on m_sampler. This creates the requested number of samples, evaluates them and stores them in the m_sample array in m_sampler.

evalLegendre(int l, int m, double x)

This method requires three arguments, l, m and x, where l is the band, m the index, and x the argument to $P_l^m(x)$. This method is an implementation of the three recurrence relation (2), (3) and (4). A pseudocode breakdown of the method follows:

```
{
    pmm set to 1
    if m>0
    {
        somx2 set to root(10x^2)
        factorial set to 1
        until band number is met
        {
            pmm * pmm(-factorial)*somx2
            factorial + 2
        }
        if l equals m
        {
                pmmp1 = x((band*2)+1)*pmm
        }
        if l equals m+1
        {
                pll = 0
                until band+2 is met
                {
                        pll=((2*currbandnum-1)*x*pmmp1-(currbandnum+numbands-
                        1)*pmm)/bandsremaining
                }
                pmm set to pmmp1
                pmmp1 set to pll
        }
        return pll
}
```

This method evaluates the Legendre polynomial using the recurring relations stated above. The order of usage of these relations was stated both by Green [GREEN03,p10] and Schönefeld, [SCHONEFELD05,p5]. This function is called by shConstruct::evalSample(), and thus is called for every sample during shSampleCollect::genSamples.

evalScaleFactor(int l, int m)

This method requires two arguments, the number of the band as l, and the index as m. It is an implementation of (6). The function is calculated and then rooted before return. It makes two calls to shConstruct ::evalFactorial at $(l-m)!$ and at $(l+m)!$. A pseudocode breakdown of this method follows:

```
{
        temp = ((2*l+1)*(factorial(l-m)))
        temp /= (4*pi*(factorial(l+m))
        root temp
        return temp
}
```

This calculates K_l^m as shown in (7). It is called in shConstruct::evalSample() in order to evaluate the scaling factor for the SH basis function $y_l^m(\theta,\varphi)$ and so is called for every sample.

evalSample(int l, int m, double theta, double phi)

This method requires four arguments, l as the index, m the band number, $theta$ and phi the spherical coordinates of the sample. It returns a coefficient for the sample as a double, which is stored in the sample itself as m_coefficient. The evaluation takes place once, either for the case where $m=0$, $m>0$, or for every other case, where $m<0$. This is an implementation of (6). A pseudocode breakdown of this method follows:

```
{
        if m=0
        {
                return root2 * scalefactor * evaluated Legendre()
        }
        if m>0
        {
                return root2 * scalefactor * cos(m*phi) * evaluated Legendre()
        }
        else
        {
                return root2 * scalefactor * sin(m*phi) * evaluated Legendre()
        }
}
```

evalSample is called from shSampleCollect::genSamples() for every sample.

evalPCFactorial(int x)

This method requires an integer argument as the factorial requested. Green suggested an array of precomputed factorials, however in this case the first 15 are precomputed; factorialArr contains 15 factorials. If the factorial requested is greater, the 15th member of factorialArr is used as a base to compute the requested factorial, which is then returned as a double. A pseudocode breakdown of this method follows:

```
{
        create factorial array
        fill with precomputed variables
        if requested factorial is less than 15
        {
                return factorial from array
        }
        else
        {
                calculate factorial using last array value as a base
                return calculated value
        }
}
```

projectFunction(hdrManip *hdr)

This method requires a hdrManip as an argument, as it collects lighting information and calculates the coefficients that will be sent to the shader for reconstruction. This function is edited from Green's project_polar_function which generates the coefficients based on the samples, the weight and the number of samples. A pseudocode breakdown of this method follows:

```
 {
        number of samples returned = bands*bands
        create local sample array
        set factor to 4*pi/number of samples
        allocate mem to m_coefficients based on number of samples returned
        create iterator
        for all samples
        {
                get colour based on direction from image
                for number of samples returned
                {
                        multiply colour by sample coefficient
                        add to coefficients vector
                }
        }
        for number of samples returned
        {
                multiply sample by factor
        }
}
```

This method implements (8). The lighting contribution is given by the colour return from the hdrManip. This is then multiplied by the sample and then added to the coefficient vector, where is it multiplied again by the $4\pi/N$, the probability factor from the Montecarlo integrator. This function is called once when the coefficients need to be calculated, during SHarmonics::begin(). It is however used again when the source image is changed by user input, during SHarmonics::input().

4.4 SHarmonics

SHarmonics allows interface between the spherical harmonics functions in shConstruct and the rest of the program, with methods for drawing and input, along with the input files required for the light term inclusion in shConstruct::projectFunction(). It stores a number of hdrManip objects which contain hdr images for input to the function projection. It also contains a number of floats and GLints used to store variables that will be sent to the shader to reconstruct the image, variables for storing user input, along with four models. It contains an a shConstruct called m_sphericalHarmonics, and an instance of a simple shader manager called m_shader.

constructor

The SHarmonics constructor requires no arguments, and initialises a few members. The hdrManips are allocated memory, m_sphericalHarmonics' constructor is called with a number of samples as an argument. A new shader program is created, and the ambient, diffuse and specular contribution multipliers, m_Ka, m_Kd and m_Ks are set, along with the default object to be shown and the default hdr map. The default number of samples is 100000.

begin()

This method initialises the SHarmonics and is designed to be called once. It loads the various files and calls the functions required to set up the spherical harmonics. It also retrieves variables from the shader and loads the relevant .obj files. A pseudocode breakdown of the method follows:

```
{
        call loadhdr function on hdr files
        set the shader files
        link shader
        retrieve Ka, Kd, Ks and gain from shader
        retrieve harmonic coefficients from the shader
        project spherical functions using default hdr image
        load obj files
        set default gain
}
```

This method is called from GLWin::begin() once, in its own initialisation loop. The default variables are set in the code itself, with Ka at 0, Kd at 1, and Ks at 0. The default image loaded is "beach_cross.hdr" by Paul Debevec [DEBEVEC], and the .obj files loaded are the

"Stanford Bunny", "Dragon", and "Happy Buddha", from the Stanford 3D Scanning Repository [STANFOBJ]. The default gain value is set to 0.3 for "beach_cross.hdr". It is changed when a new image is selected, but can also be changed by the user.

draw()

This method requires no arguments and draws the objects requested. It also sets shader variables prior to drawing the object. A pseudocode breakdown of the method follows:

```
{
        glclear
        set shader to use
        set Ka, Kd and Ks on shader
        create local coefficient array
        set harmonic coefficients on shader
        set gain on shader
        switch based on selected obj
        {
                1: translate and scale
                draw bunny

                2: translate and scale
                draw dragon

                3: translate and scale
                draw buddha

                4: translate and scale
                draw sphere
        }
}
```

This method is called whenever the drawing of the object needs to be updated.

input(unsigned char key)

This method requires a key input on which a switch statement is performed. A pseudocode breakdown of this method follows:

```
{
        switch on keypress
        {
                if(1/2/3/4)
                {
                        change load number (affects model)
                        call draw()
                }
                if(, or .)
                {
                        change gain
                }
                if(e/r/t)
                {
                        change sh number (affects image)
                        reproject spherical functions
                        change gain depending on image
                        call draw()
                }
        }
}
```

This is a simple method that is called whenever a key input is sent to GLWin::input(). The gain per image can be modified for aesthetic reasons, using the "," and "." keys.

4.5 hdrManip

hdrManip is a class used for storing and loading a hdr image in vertical cross format. This class uses the RGBE header file written by Bruce Walter which allows reading of the RGBE file format developed by Greg Ward, hosted at [CORNELL].

The hdrManip class contains an image array, which consists of a series of RGB float values, with array[0] as red, array[1] as green, and array[2] as blue. This is created by RGBE::ReadPixels_RLE, which is called by hdrManip::ReadImageRGBE. The class also stores the height and width of the image loaded as integers.

constructor

The constructor requires no arguments, and sets the image array to 0, as well as the height and width members.

returnColour(ngl::Vector vectorIn)

This method requires an input vector as an argument. This is called by shConstruct::projectFunction to retrieve the colour of the pixel in the image based on the direction of the vector which has been previously converted from spherical coordinates into a Cartesian vector. A pseudocode breakdown of the method follows:

```
{
        declare local vector equal to input vector
        normalise direction vector
        set 1/pi
        set invl=1/root(x component squared + y component squared)
        set radius to 1/pi * acosf( z component) * invl
        set u to x component * radius
        set v to y component * radius
        set x = u * width squared
        set y = v * height squared
        set index to y * width + x * 3
        red component = imagearray[index]
        green component = imagearray[index+1]
        blue component = imagearray[index+2]
        return r,g,b as an ngl::Vector
}
```

This method finds the RGB component on the image of the location of the spherical coordinate that is being sampled in shConstruct::projectFunction.

readImageRGBE(char fileIn)

This method requires the location of a file as input. It opens a file for reading, clears the current image array, and writes pixels to sets of three floats as RGB data, using a method contained in the RGBE header by Bruce Walter. A pseudocode breakdown of this method follows:

```
{
        create rgbe_header_info
        open file
        delete image array
        allocate space for the image array
        set image array to 0
        call rgbe_readpixels_RLE from RGBE header
        close file
}
```

This method is called during SHarmonics::begin() where three images are loaded.

4.6 shaderProgram

shaderProgram handles the loading and usage of a shader. It was created with the intention of providing a lightweight interface to send and receive variables to and from a shader. It was based off a blog post by Jon Macey entitled "GLSL Shader Manager design Part 2" [MACEY10]. It contains a number of methods used for loading, compiling, attaching and linking a shader.

constructor

The constructor for shaderProgram requires no arguments and creates a shader program under m_program.

setShader(char inFrag, char inVert)

This method requires the location of both the fragment shader and the vertex shader as arguments. It relies on a number of GL calls in order to read, compile and attach the shader. A pseudocode breakdown of this method follows:

```
{
        create fragment shader
        create vertex shader
        read infrag into a char array
        read invert into a char array
        compile fragment shader
        compile vertex shader
        get compile status for fragment shader, exit if error
        get compile status for vertex shader, exit if error
        attach fragment shader to program
        attach vertex shader to program
}
```

Errors are output if either the fragment or the vertex shader fail to compile. This method is called by SHarmonics::begin(), whereupon it loads "shaders/reconstruct.fs" as the fragment shader and "shaders/reconstruct.vs" as the vertex shader.

link()

This method links the shader. It was developed with code from [MACEY10]. A pseudocode breakdown of this method follows:

```
{
        create information buffer
        link program
        get link information
        if information exists
        {
                get error message
                cerr the error message
                delete message
                cerr the link failure
```

```
            exit
        }
}
```

This method returns errors if they are found during the linking process, and causes the program to exit if they are. It is called once from SHarmonics::begin().

readShaderFile(char inFile)

This method is used within setShader to read a file into a character buffer. It is called twice, once for the fragment and once for the vertex shader. It will inform the user with an error if it cannot open the specified file. It returns a character array, which within the method is called charBuffer.

readFileSize(char inFile)

This method returns the size of the file in bytes. It is called from readShaderFile() so that the character array charBuffer is of the correct size. <sys.types.h> and <sys/stat.h> are used, as filestatus.st_size returns the size of a file in bytes. Another possible option would have been to read the file and count the number of bytes. This method returns the size in bytes as an integer.

4.7 GLWin

GLWin handles initialisation, drawing, resizing and input for the program. During begin(), the GL initialisation occurs, and a SHarmonics m_sh is created. At this point, SHarmonics.begin() is called, and the application is ready to render an object lit using spherical harmonic based lighting.

begin()

This method initialises GL to its current state, and creates and initialises m_sh which is an object of class SHarmonics.

resize()

This method allows for resizing of the GL window.

paint()

This is the draw method of GLWin, and handles top level transformations affected by user input. Once the transformation is complete, draw is called from m_sh.

input()

This method sends keystrokes to SHarmonics::input(), as well as detecting and switching on the input keys in order to allow the user some simplistic movement around the object. The input from these six keys are taken into account in paint() and occur before m_sh is drawn.

4.8 Main

Main.cpp contains a MainWindow, which houses only a GLWin called m_window, and calls the various functions initialize(), anim(), reshape(), display() and input() from GLWin.

4.9 SH Creation and display

The following pseudocode shows the process undertaken to create and display an object lit via spherical harmonic lighting in this program.

```
{
        create SHarmonics object
        {
                add new hdrManips
                create a new shConstruct
                {
                        create new samples based on input value
                        for each sample
                        {
                                convert spherical coord to cartesian
                                evaluate sample
                                store coefficient in sample
                        }
                }
                create new shaderprogram
                set shader variables ka, kd, and ks
                set user input variables
                load images to hdrManips
                {
                        read headers
                        read file and store rgb values
                }
                set shader files
                link shader
                retrieve current ka, kd, ks and gain values from shader
                retrieve current harmonic coefficients from shader
                project spherical functions into selected image
                {
                        init base variables
                        create local sample array
                        for numsamples
                        {
                                for numsamples in retrieval
                                {
                                        retrieve colour from image for spherical coordinate
                                        multiply colour by sample coefficient
                                        add result to coefficients array
                                }
                        }
                        multiply every coefficient in array by factor
                }
                load obj files
                set base gain for current image
                send current ka, kd and ks to shader
                create local coefficient array
                send harmonic coefficients to shader
                send gain to shader
                scale, translate and draw the selected model
                wait for user input and redraw

}
```

Reconstruction of the image on the shader

The shader handles the reconstruction of the image from the SH coefficients passed to it. This is based on the reconstruction given by Ramamoorthi et al in (9). The shader in use is based off a shader at Rendering Wonders [RWND] which in turn is based off Ramamoorthi's design (9). The shader requires a number of variables, such as the harmonic coefficients and gain. It also allows for control of Ka, Kd and Ks but these are not implemented as yet.

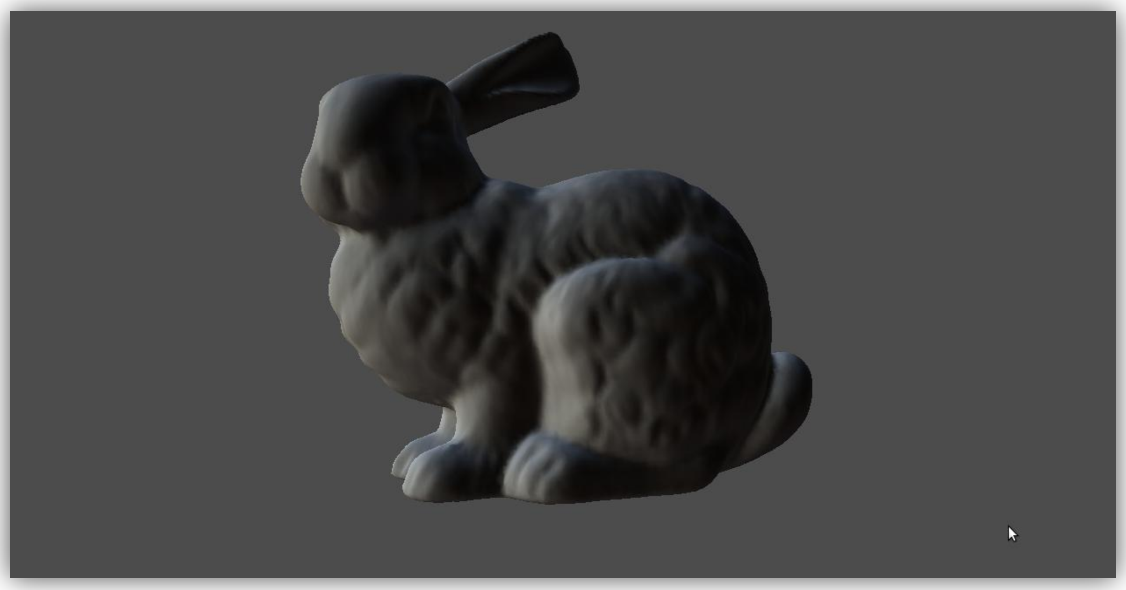

Figure 4.2: Rendering example: The Stanford Bunny lit by spherical harmonics based lighting. The light probe used is Paul Debevec's "beach_cross". 100000 samples and 3 bands were used.

5.0 Results

The application created does render images based on a number of vertical cross format .hdr files courtesy of Paul Debevec. The spherical harmonics return different results when different sample numbers are used, as this directly affects the accuracy of the reproduction of the input image, as can be seen in Figures 5.1 to 5.4.

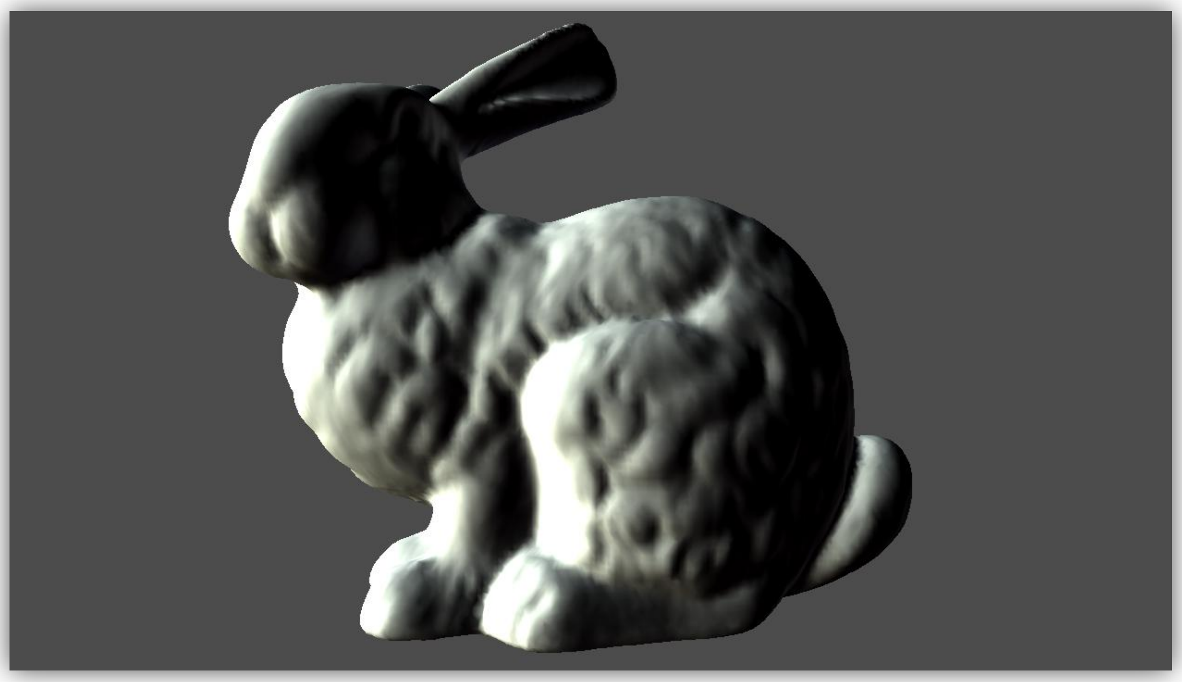
Figure 5.1: The Stanford Bunny rendered with only 10 samples.

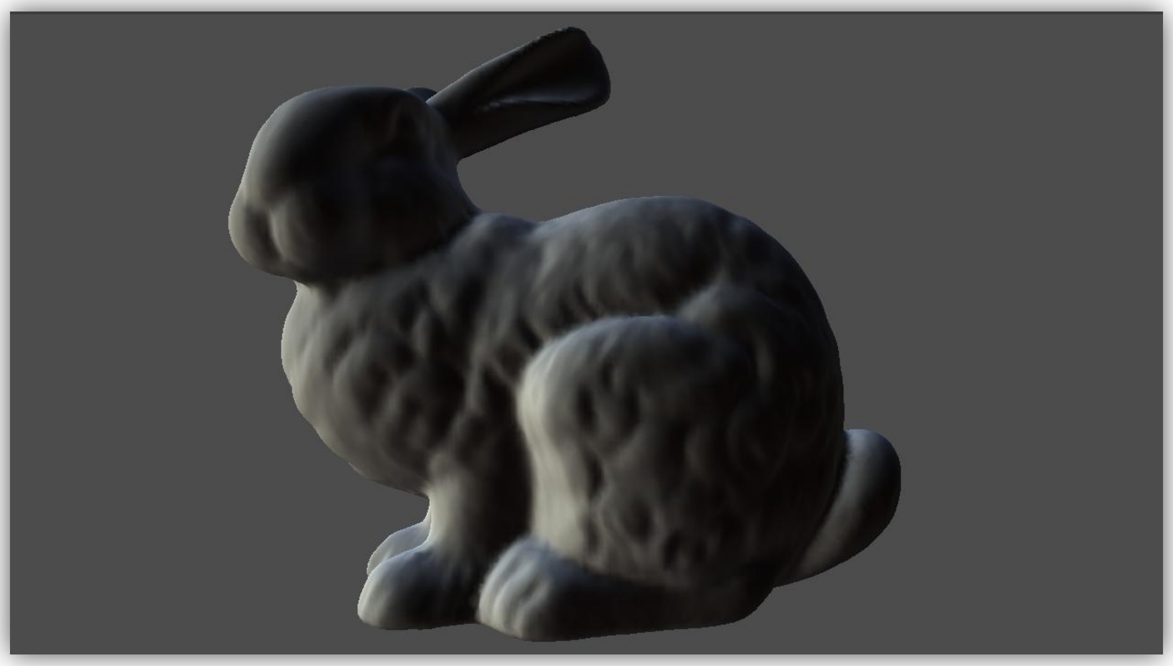
Figure 5.2: The Stanford Bunny rendered with 100 samples.

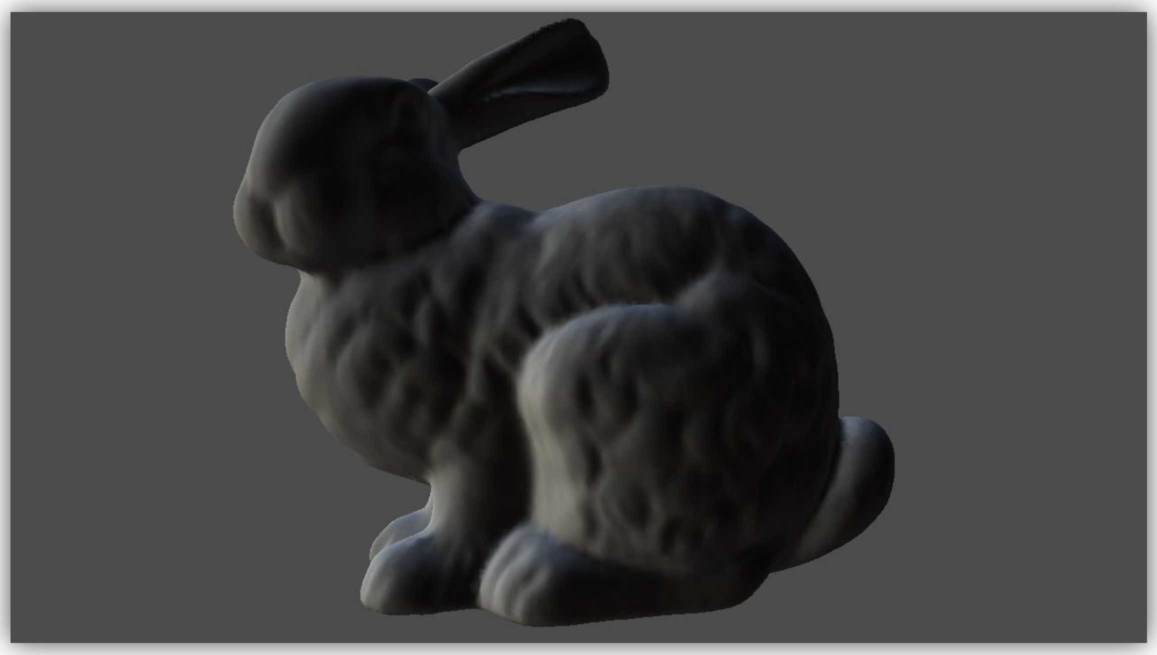
Figure 5.3: The Stanford Bunny rendered with 1000 samples.

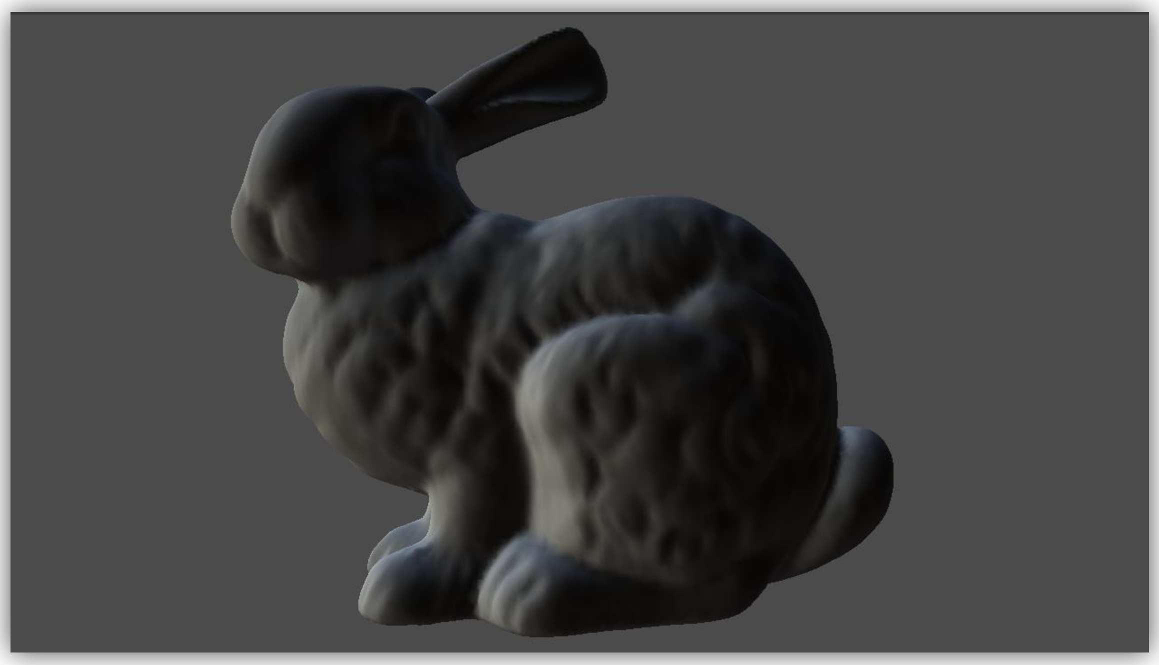
Figure 5.4: The Stanford Bunny rendered with 10,000 samples.

The default number of samples is 100,000 as there does not appear to be any noticeable change in load or run speed. The quality of the lighting however still depends on the quality of the input image. Each image has a different tolerance for the number of samples required before image quality noticeably decreases, and this varies depending on the frequency of unique light directions in the original image.

The number of bands affects the quality of the image as it limits the number of coefficients available. The term "band" is interchangeable with the "order" of harmonics. Figure 5.5 shows image created with only one band. Image quality increases along with the number of bands, due to the number of resulting coefficients, though this is merely a margin of accuracy. This implementation requires only three bands for the resulting nine coefficients. Image quality will noticeably be adversely affected by a number of bands lower than three.

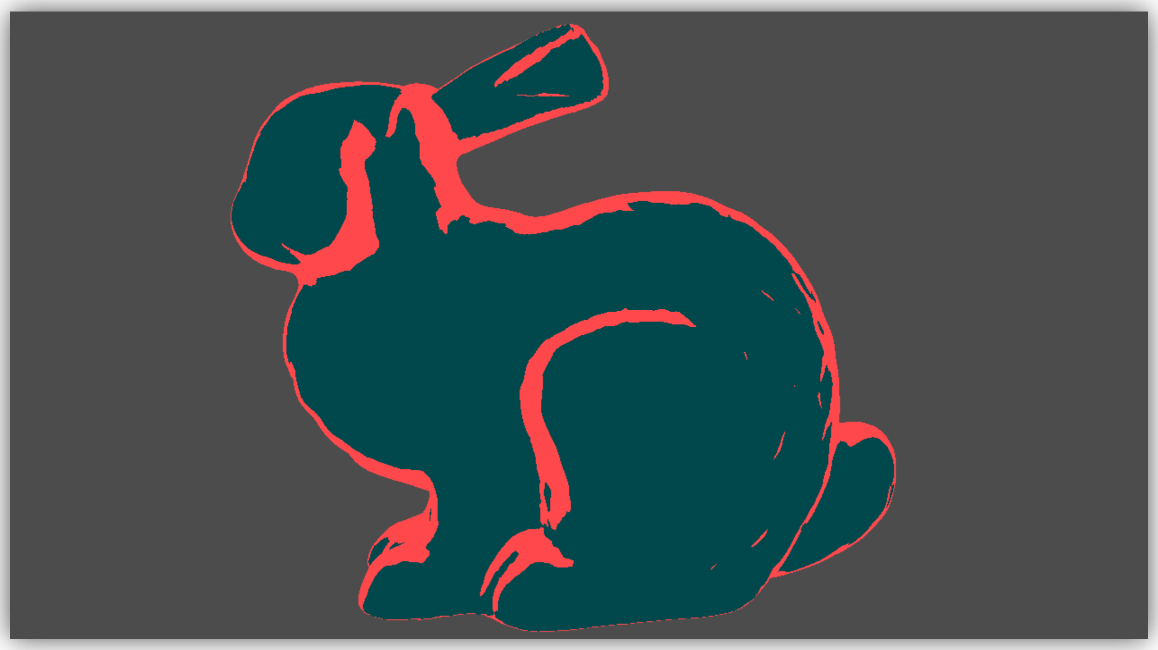
Figure 5.5: The Stanford Bunny rendered with 10000 samples but only one band.

As can be seen in figure 5.5, shading errors appear as there is not enough information to properly reconstruct the surface. Errors also appear in figure 5.6 with a large yellow contribution appearing from an incorrect direction.

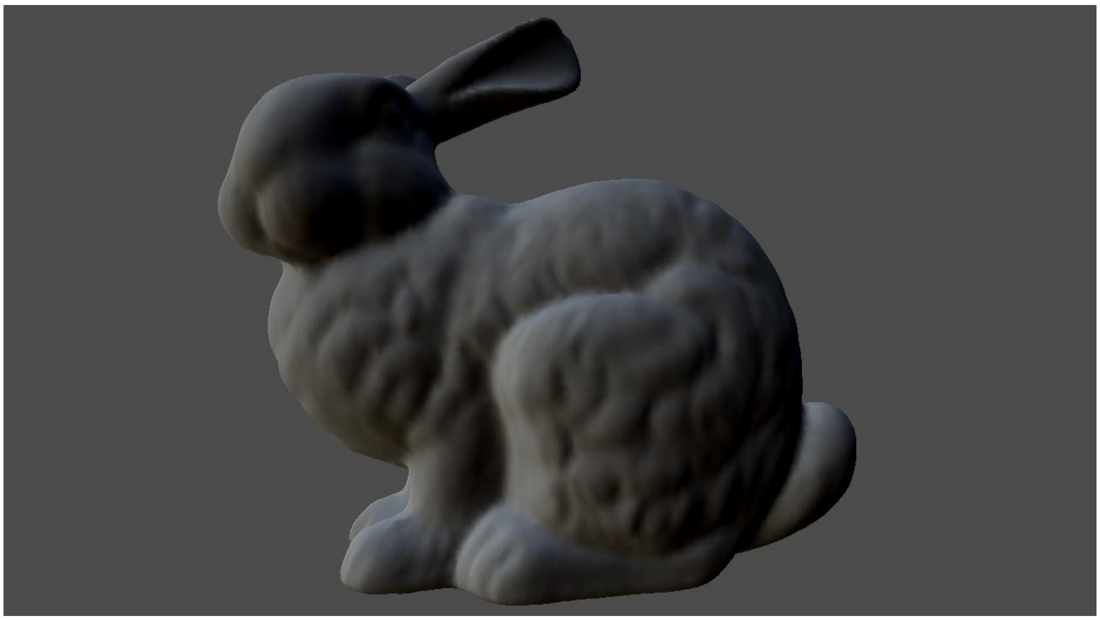
Figure 5.6: The Stanford Bunny rendered with 10000 samples and only 2 bands.

This application runs at 620FPS on an Intel Qx6850 clocked at 4.1Ghz with an nVidia GTX280 at stock clocks which shows that the application does provide real-time SH-based lighting.

6.0 Conclusion

The aim of this project was to light and render an object using spherical harmonics based lighting. As a result it could be concluded that this project was successful, but with much room for improvement.

The produced application has a number of issues; there is no mouse control of the object, no user interface, and it is built with GLut. The application could be converted to a Qt GUI application and built using <QtOpenGL>, with a gui for loading .hdr files and controlling input. The application itself still uses some depreciated code, which could be remedied.

The implementation of the spherical harmonics system itself is extensible. It could be easily expanded to return more than nine coefficients, as it is already able to create more than 3 bands. Whilst the sampling is relatively fast, the ability to sort the samples by "importance" could allow for more samples where they are needed, and less where there is not much variation in the environment light intensity. Another possible extension would be to use a number of probes for a room and interpolate between "cells", similar to the method used by Halo 3, which would allow the object to be moved though a SH based lighting environment as opposed to the static light that is currently rendered.

Spherical harmonics rotations are not currently implemented, though this could be achieved using the ngl::Matrix class and the method outlined by both Schönefeld and Green [SCHONEFELD05] and [GREEN03]. This would allow for a dynamic "night and day" procedural lighting implementation, which could then be rotated to allow for a spherical harmonics-based sky system. This could be further extended by layering a "sun" light into the samples, allowing for a sun and sky system to be created.

Spherical harmonics allow fast computation of a number of processes that are useful in games, such as interreflection, ambient occlusion, and soft shadowing. Unfortunately, these processes require raytracing as outlined in [GREEN03]. The time allocated has been spent on the implementation of the spherical harmonics and not a raytracer. Were a raycaster to be implemented, it would allow for the real-time calculation of soft shadows, interreflections and non-screen space ambient occlusion.

During this project, knowledge has been acquired which allows the theory behind spherical harmonics and the mathematics required for their implementation to be understood, and resulted in an enjoyable learning experience.

7.0 References

[SAINDON] VFX Supervisor Eric Saindon in an interview with www.cgsociety.org, page 2. Accessed at http://www.cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/avatar on 07/2011.

[GUERRILLA07] Guerilla Games, Deferred Rendering in Killzone 2, Accessed at http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf on 07/2011.

[IGNKZ2] Image courtesy IGN Entertainment, Accessed at http://ps3media.ign.com/ps3/image/article/972/972576/killzone-2-20090414092626423.jpg on 07/2011.

[CHEN08] Hao Chen, Microsoft Research Asia, Accessed at http://www.bungie.net/images/Inside/publications/presentations/lighting_material.zip on 07/2011.

[UDN11] Unreal Developer Network, Development Kit Build Upgrade Notes, Accessed at http://udn.epicgames.com/Three/DevelopmentKitBuildUpgradeNotes.html on 07/2011

[WFIRE11] Wolfire Blog, GDC session summary: Battlefield 3 Radiosity, Accessed at http://blog.wolfire.com/2011/03/GDC-session-summary-Battlefield-3-Radiosity on 07/2011

[EINARSS10] Per Einarsson, A Real Time Radiosity Architecture for Video Games, SIGGRAPH 2010, Accessed at http://advances.realtimerendering.com/s2010/Martin-Einarsson-RadiosityArchitecture(SIGGRAPH%202010%20Advanced%20RealTime%20Rendering%20Course).pdf on 07/2011.

[SLOAN02] SloanP. P., Kautz J., and Snyder J. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments."SIGGRAPH 2002, Computer Graphics Proceedings, p527–536, 2002.

[RAMAMOOR01] Ravi Ramamoorthi and Pat Hanrahan, "An efficient representation for irradiance environment-maps", SIGGRAPH 2001, Los Angeles, CA, p497-500, 2001.

[SCHONEFELD05] Volker Schönefeld, "Spherical Harmonics". Accessed at http://heim.c-otto.de/~volker/prosem_paper.pdf on 07/2011.

[GREEN03] Robin Green, "Spherical Harmonic Lighting: The Gritty Details", Game Developers Conference 2003. Available at http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf , Accessed 07/2011

[DEBEVEC] Paul Debevec, "Light Probe Image Gallery", Accessed at http://ict.debevec.org/~debevec/Probes/ on 07/2011.

[STANFOBJ] Stanford University, "Stanford 3D Scanning Repository", Accessed at graphics.stanford.edu/data/3Dscanrep/ on 07/2011.

[CORNELL] Bruce Walter, RGBE header, Accessed at http://www.graphics.cornell.edu/~bjw/ on 07/2011.

[MACEY10] Jon Macey, "GLSL Shader Manager design Part 2", Accessed at http://jonmacey.blogspot.com/2010/11/glsl-shader-manager-design-part-2.html on 07/2011.

[RWND] Marc Costa, "Spherical Harmonics", Accessed at http://renderingwonders.wordpress.com/2011/05/28/spherical-harmonics/ on 07/2011.

[CPROJECT] Karsten Schwenk, "A Simple Class for Spherical Harmonic Expansion", Accessed at http://www.codeproject.com/KB/cpp/sh_projection.aspx on 07/2011.

USER'S GUIDE

Once the application is compiled, there are a few controls once it has loaded. The application takes a few seconds to begin, primarily due to .obj loading.

Controls:

Q	move: -Y
Z	move: +Y
W	move: -Z
A	move: -X
S	move: +Z
D	move: +X
1	model: bunny
2	model: dragon
3	model: buddha
4	model: sphere
E	image: beach
R	image: uffizi
T	image: hotel
,	decrease gain
.	increase gain
ESC	exit application