GPGPU Accelerated Texture-Based Radiosity

Sungkono Surya Tjahyono, Christof Lutteroth, Burkhard Wünsche
Department of Computer Science
University of Auckland
Auckland, New Zealand
Email: stja001@aucklanduni.ac.nz, c.lutteroth@auckland.ac.nz, b.wunsche@auckland.ac.nz

***Abstract*—Radiosity is a popular global illumination algorithm capable of achieving photorealistic rendering results. However, its use in interactive environments is limited by its computational complexity. This paper presents a GPGPU-based implementation of the gathering radiosity approach using texture-based discretisation and the OpenCL framework. Hemicubes are rendered to a texture array and processed by OpenCL kernels in parallel to compute the output radiance of the patches. Results show that even with the high synchronisation overhead of the OpenGL-OpenCL interoperability, the proposed method is an order of magnitude faster than a CPU-based implementation, and that it approaches interactive speeds. Investigation of the influence of different parameters shows that an increase in hemicube size results in a linear increase in computation time, while an increase in the number of layers in the texture array dimensions results in a logarithmic decrease in computation time.**

I. Introduction

Radiosity is a Global Illumination (GI) algorithm which was proposed in 1984 [1]. The method solves the rendering equation [2] by using a finite element discretisation to calculate light interaction between diffuse surfaces. Radiosity can be precalculated and stored as lightmaps as long as the lighting condition and/or geometries within the scene do not change. Once one of the assumed conditions is changed, the calculated radiosity solution is no longer valid and it has to be recalculated.

Typical radiosity implementations work by subdividing the polygons within the scene into smaller elements (usually referred to as patches) and light interactions are performed at this level. The geometric relationship between a pair of patches is referred to as form factor. The first implemented radiosity solution had $O(n^2)$ complexity for its memory and computation requirements, since the interaction between all patches had to be taken into account.

There have been many approaches to eliminate these constraints, including the progressive refinement approach (shooting and gathering) [3] and instant radiosity [4]. The shooting approach works by choosing the patch with the highest undistributed energy and shooting it to all the visible patches and the process is repeated until the highest undistributed energy falls below a certain threshold. The gathering approach works by going through each patch sequentially and summing up the irradiance energy coming from all visible patches and this process is repeated until the intensity difference between iterations fall below a certain threshold. Instant radiosity works by shooting rays from the light sources in random directions and places a Virtual Point Light (VPL) at the point of intersection between the rays and surfaces in the scene. The VPL's intensity is calculated with the value of the colour of the light source and the reflectance properties (albedo) of the surface it hits. A Russian Roulette scheme is used to determine whether new rays should be cast from the VPL or not. The instant radiosity approach is view dependent, unlike the other two approaches.

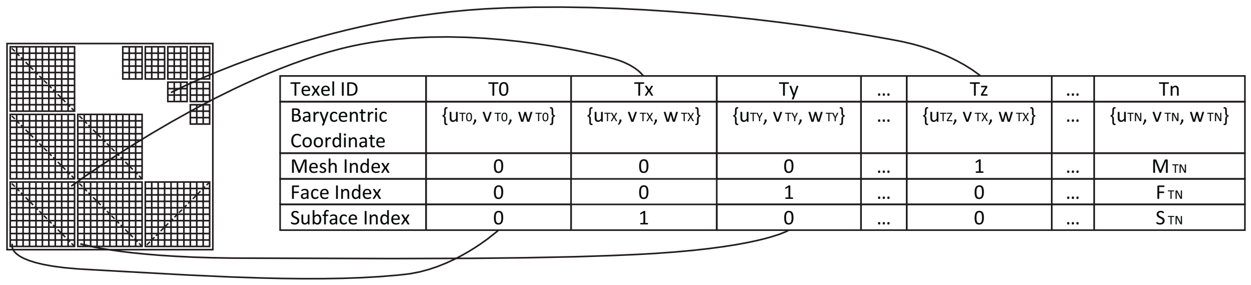
The progressive refinement approach usually uses a technique called the hemicube rendering [5] to obtain a visibility list for each patch. This list can be used to check for visibility and to calculate form factors faster than other methods such as ray casting but suffers from aliasing and banding artifacts due to the limited resolution of the hemicube.

Even with the current generation of powerful consumer level hardware, calculating the radiosity solution in real time is still a challenge, regardless of which approach is taken. Motivated by the advent of highly programmable and parallelisable graphics hardware (GPU) and their supporting frameworks (NVIDIA CUDA, ATI Stream, Khronos OpenCL, Microsoft DirectCompute), this research looks at exploiting the highly parallel nature of the radiosity calculation by using the GPU to process most or all of the lighting data. In this paper, a novel method is proposed to solve the radiosity equation (gathering approach specifically) on the GPU by representing the radiosity patches in the texture space. In addition, GPU-based computation using the OpenCL framework is compared to CPU-based computation using a sample scene to evaluate their performance. Although the GPGPU frameworks have been around for quite some time, its techniques are still not as matured as shader-based techniques and published radiosity techniques using the GPGPU frameworks are still scarce.

The rest of the paper is organised as follows: Section II presents related works in the area of radiosity. Details of the proposed method are described in Section III. Section IV outlines important implementation details with regards to OpenGL-OpenCL interoperability. An evaluation of the implemented system is presented in Section V. Limitations and ideas for future work are presented in Section VI. The conclusion is outlined in Section VII.

II. Related Works

There have been many attempts to calculate radiosity in real time and some have been very promising. Nielsen et



Texel ID	T0	Tx	Ty	...	Tz	...	Tn
Barycentric Coordinate	$\{u_{T0}, v_{T0}, w_{T0}\}$	$\{u_{TX}, v_{TX}, w_{TX}\}$	$\{u_{TY}, v_{TY}, w_{TY}\}$	...	$\{u_{TZ}, v_{TX}, w_{TX}\}$	...	$\{u_{TN}, v_{TN}, w_{TN}\}$
Mesh Index	0	0	0	...	1	...	M_{TN}
Face Index	0	0	1	...	0	...	F_{TN}
Subface Index	0	1	0	...	0	...	S_{TN}

Fig. 1. UV Coordinate Set (left) and the *patch list* extracted from texture space (right).

al. [6] use texture space subdivision and hardware texture mapping to accelerate the hemicube rendering process. An index map (integer values encoded as colour values) is used to represent radiosity patches and is texture mapped onto the polygons when rendering the hemicubes. The rendered values on the hemicube are used as index values to a form factor table containing information such as form factor, surface id and element index. Coombe et al. [7] also use texels as radiosity patches and use a modified progressive refinement approach that runs on the GPU. A hemicube is rendered from the point of view of the shooter as normal, but instead of shooting energy to all patches in the hemicube, all patches are iterated over in a fragment shader to check if they are visible from the shooter. If a patch is visible, it receives a fraction of the shooter's energy based on its form factor. Going through the list of potential receivers rather than going through all the patches in the hemicube eliminates the problem of having to write to arbitrary locations in the textures. It also allows the shooter's energy to be distributed in parallel since patches are independent of each other.

In the game industry, the most notable success in real time radiosity calculation is a product called Enlighten by Geomerics [8]. It separates the direct and indirect lighting and blends the output of each at the final stage of rendering. This is done to allow for the direct lighting effects, which are generally high frequency and instantaneous, to be displayed as they happen, and the indirect lighting effects, which are generally soft, subtle and low frequency, to be integrated over time. Separating the direct and indirect lighting solution also allows Enlighten to be plugged-in into an existing lighting solution as long as it can generate the appropriate input for Enlighten to process. Geomerics has adopted the NVIDIA CUDA framework to improve the performance of Enlighten's preprocessing and runtime components, resulting in a complete update of a game level to take 2-3ms [9].

Castaño [10] renders low resolution hemicubes to a texture atlas and multiplies them with a multiplier map (Fig.2) which encodes the amount of light received per solid angle of the texels of the hemicube according to Lambert's emission law. This is then integrated to obtain the output radiance value of each patch. By using the mulitplier map, the form factors do not need to be calculated explicitly. The Lambert's emission law map takes into account the angle between the surface normal of the patch on which the hemicube is rendered from and the texel on the surface of the hemicube, while the solid angle map takes into account the texel's orientation, distance and position on the hemicube. The multiplier map can be calculated once and be applied to all hemicubes. Castaño uses geometric subdivision with irradiance caching to allow for less hemicubes to be rendered and to give smoother appearance in the rendered scene.

III. Proposed Method

A. Texture space subdivision

Most geometry-based subdivision techniques in radiosity systems use quadrilaterals (quads). In the texture space, texels serve as a good candidate to represent such a subdivision. Instead of subdividing the quads geometrically, a UV coordinate value is generated for each vertex. These UV coordinate values are also used to create lightmaps which are texture mapped onto the scene when rendering the hemicubes and displaying the converged radiosity solution.

As shown in Fig.1 five sets of data need to be extracted from the texture space to create the *patch list*. These are:

- *texel id* of the texel within the texture.
- *mesh index* of the mesh the texel belongs to.
- *face index* of the face within the mesh identified by *mesh index*.
- *subface index* of the subface within the face identified by *face index*.
- *barycentric coordinate* of the center of the patch.

The *face index* and *subface index* are used to differentiate between the quad that makes up the surface and the triangles that make up the quad. In the case that the surface is a triangle, there will only be a single subface. This is done so that the rendering system only needs to deal with one type of geometry and it also allows for the calculation of the barycentric coordinates. The barycentric coordinate is calculated by weighting the position of the center of the texel with regards to the three vertices that make up the triangle. The barycentric coordinate can be used to calculate the center and normal of the patch by substituting the UV coordinate of the vertices with the normal and position of the vertices in the world coordinate space.

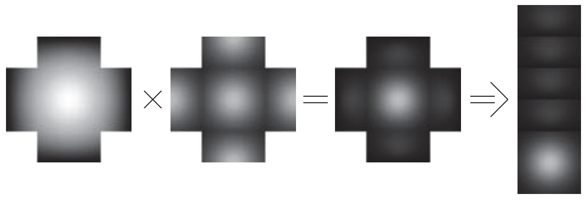


Fig. 2. Lambert's emission law map (left), solid angle map (inner left), multiplier map (inner right) and stacked format multiplier map (right). The solid angle and multiplier maps have been intensified to show the distribution of values across the maps more clearly.

B. Hemicube

A hemicube is used to obtain the visibility list and the current radiosity values of the patches in the scene (values on the lightmap). Hemispherical projection cannot be used in this method due to the low geometric subdivision, as all of the details are encoded in the texture space. As mentioned by Elias [11] and Castaño [10], instead of calculating the form factors between patches, a multiplier map (Fig. 2) can be applied to the hemicubes to obtain the contribution from each texel in the hemicube towards the irradiance value.

IV. Implementation Details

A. Patch List

The *patch list* (Fig.1) can be obtained by using a combination of the render to texture technique (RTT), two floating point RGBA render targets, a vertex shader and a fragment shader. The vertex shader transforms the UV coordinates using an orthographic projection, while the fragment shader calculates the barycentric coordinates and stores them in the first render target. The mesh, face and subface indices which are passed from the host application as vertex attributes are stored in the second render target by the fragment shader. The alpha channel of each render target is used to store a flag to indicate that the texels are valid and have been rasterised by the fragment shader.

The two render targets are then transferred back to the host application, where each RGBA tuple is processed to extract the information from all valid tuples (the flag in the A component is set). This step can be performed as an offline process or as an initialisation step depending on the complexity of the scene. The *patch list* can be exported to a binary file as all the information contained within it is static.

B. Hemicube Rendering

To reduce the rendering time as much as possible and to provide enough workload for the GPU, the hemicubes are rendered to a 2D texture array using a geometry shader and an instancing technique. By using these techniques, the same hemicube face can be rendered to multiple layers of the texture array with a single draw call. An OpenGL extension called `GL_ARB_viewport_array` can also be used to allow for multiple viewports and scissor rectangles to be specified, allowing for a complete hemicube to be rendered with a single draw call. By combining these three techniques, a single draw call can render complete hemicubes to multiple layers of the texture array to reduce the CPU workload. The geometry shader takes as input a list of matrices which represent the model-view projection for each face of the hemicube, while the viewport and scissor rectangle arrays specify which region of the texture array to draw into.

To save texture space, the hemicube is arranged not in the typical cross layout but in a vertical stack format shown in Fig.2. A random rotation ($[-180°, 180°)$) along the patch's normal is applied before the patch's hemicube is rendered to reduce the effect of banding artifacts. The banding artifacts are caused by the limited resolution of the hemicube to sample the irradiance of the environment. By adding a random rotation to each patch's hemicube, banding artifacts are traded for noise artifacts which can be smoothed out using interpolation [10]. The random rotation for each patch should be constant throughout the lifetime of the application to prevent flickering artifacts on the displayed result as the iterations converge to the true solution.

C. Hemicube Integration

OpenCL is used to perform the hemicube integration because of the highly parallelisable nature of the problem. OpenCL buffers are created from OpenGL textures using OpenCL's interoperability feature to avoid PCI Express bus transfer of the texture array to the host application and back to the GPU after being converted to OpenCL buffers. Three OpenCL kernels are used: *multiply kernel* to multiply the hemicubes with the multiplier map, *reduce row kernel* to reduce the rows of each hemicube into a single row, and *reduce column kernel* to reduce the hemicube columns into a single value.

OpenGL-OpenCL interoperability in the OpenCL 1.0 specification relies on the use of `glFinish()` and `clFinish()` to ensure synchronisation of the buffers before they can be used by the other API. As mentioned by Hensley et al. [12], `glFinish()` and `clFinish()` are heavyweight, expensive and blocking calls. The OpenCL 1.1 specification adds two event extensions (`cl_khr_gl_event` and `GL_ARB_cl_event`) that allow for easier OpenGL-OpenCL buffers synchronisation. These events are much more lightweight and should reduce the amount of overhead. Using this event mechanism, an event can be placed in the OpenGL command queue and checked by OpenCL for synchronisation before trying to acquire the OpenGL buffers. Once the event is processed by the OpenGL command queue, OpenCL can attempt to acquire the buffers without having to wait until the OpenGL command queue is empty.

The number of work items per work group to process the 2D texture array can be tweaked to obtain the highest occupancy and utilisation of the GPU. For the first two kernels, the number of work items is made as small as possible to allow for the most number of work groups. In this case, each work group contains 512 work items laid out in a 32x16x1 block (the number of maximum work items per work group

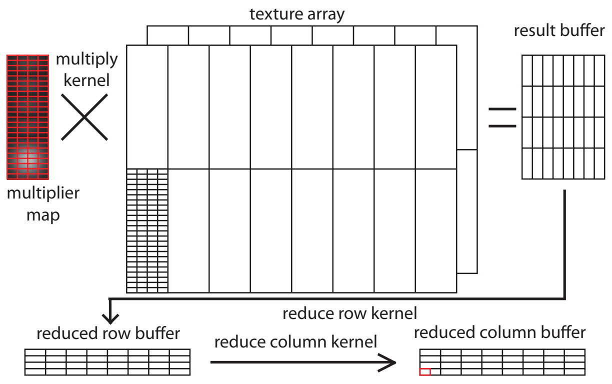


Fig. 3. The *multiply kernel* applies the multiplier map (top left) to the texture array (top middle) in parallel to produce the result buffer (top right). The *reduce row kernel* reduces all rows in each hemicube to a single row (bottom left) and the *reduce column kernel* reduces the columns to a single value stored in the same buffer (bottom right).

is governed by the available resources of the GPU and the resources used by each kernel). This means each work group is responsible for a small portion of the texture array in any one layer (each red block in the top left of Fig.3 is a work group). This gives the GPU enough workload at a fine-grained level and helps the hardware scheduler to schedule different work groups as data are fetched from the GPU's global memory to each work group's shared local memory. The *multiply kernel* converts the 3D workspace into a 2D workspace by storing subsequent layers on top of the previous layer (top right of Fig.3). This simplifies the indexing complexity within the *reduce row kernel*.

The *reduce row kernel* is responsible for reducing each column of the hemicube into a single value. In the example shown in Fig.3, if each hemicube is 128x384 and the texture array is 1024x768x2, this means the *reduce row kernel* will produce an output buffer with dimensions of 1024x4. At this stage, it is important to make sure that each work group only processes a region that belongs to the same hemicube. The *reduce column kernel* then sums each of the 128 elements into a single value by halving the number of elements in each iteration (parallel reduction). The final value from each work group is stored in the same buffer and stored at their corresponding group index so that only the first 32 elements need to be transferred back to the host application (red block in the bottom right of Fig.3). These reduced values are multiplied by the surface reflectance of the patch and stored in the radiosity texture to be used in the next iteration and when displaying the result on screen.

V. Evaluation

A. Performance

The sample scene is tested using an Intel Core i5 750 at 2.67GHz and an NVIDIA GeForce GTX 550 Ti with 1GB GDDR5 memory, 192 shader cores, compute capability 2.1, OpenGL 3.3.0, GLSL 3.30 and OpenCL 1.0 specifications. Key factors contributing to the overall performance are: hemicube resolution, lightmap resolution and texture array dimensions. The smaller the hemicube resolution and the higher the texture array dimensions are, the more hemicubes that can fit into the texture array and therefore more can be processed in parallel. The higher the lightmap resolution is, the more patches to process which reduces the performance. There are 10700 texels used to represent the radiosity patches in the sample scene on a 128x128 lightmap texture.

As shown in Fig.4, the average iteration time (per frame) is quite high. This is mainly caused by the sharing of OpenGL texture buffers with OpenCL. By sharing OpenGL buffers with OpenCL, the need to transfer the data to the host application memory and back to the GPU memory as OpenCL buffers is avoided. However, there is still some synchronisation that needs to be completed before OpenCL can use the buffers. This synchronisation is the acquiring and releasing of the buffers from one API to the other through `glFinish()` and `clFinish()` (Section IV-C). These overheads can be up to 21.2% and 60.2% of the iteration time for `glFinish()` and `clFinish()` respectively. The event based mechanism supported by the OpenCL 1.1 specifications is not implemented due to the lack of support of the necessary extensions by the NVIDIA driver at this time.

The average kernel execution time is quite stable within the different hemicube sizes and texture array dimensions. This shows that the size of the workload for the same hemicube size is constant, the main difference being how many hemicubes are processed simultaneously based on the number of layers in the texture array.

The average pixel difference graph in Fig.4 compares two radiosity techniques: the full matrix approach using a Gauss-Seidel linear solver with form factors and the proposed method. The value for each iteration represents the average difference in pixel intensity values for the 10700 texels within the lightmap textures of the two methods. It shows that there is a consistent difference in average intensity values across corresponding pixel locations between the two methods' lightmap textures across the different hemicube sizes (the three line graphs fall on top of each other). It also shows that the radiosity calculations converge to the true solutions as more iterations are performed and that the proposed method behaves similarly to the Gauss-Seidel method.

TABLE I compares the average iteration time (over 30 iterations) of a CPU-based and GPU-based implementation of the proposed method on different hemicube sizes using a 1024x1024 texture array with different number of layers. The high iteration times of the CPU-based implementation is mainly attributed to the necessity to transfer the texture array data to the application's memory and the fact that the hemicubes within the texture array are processed sequentially.

B. Artifacts

Fig.5 shows the effects of hemicube resolution and random rotation on the converged results. Banding artifacts are not as

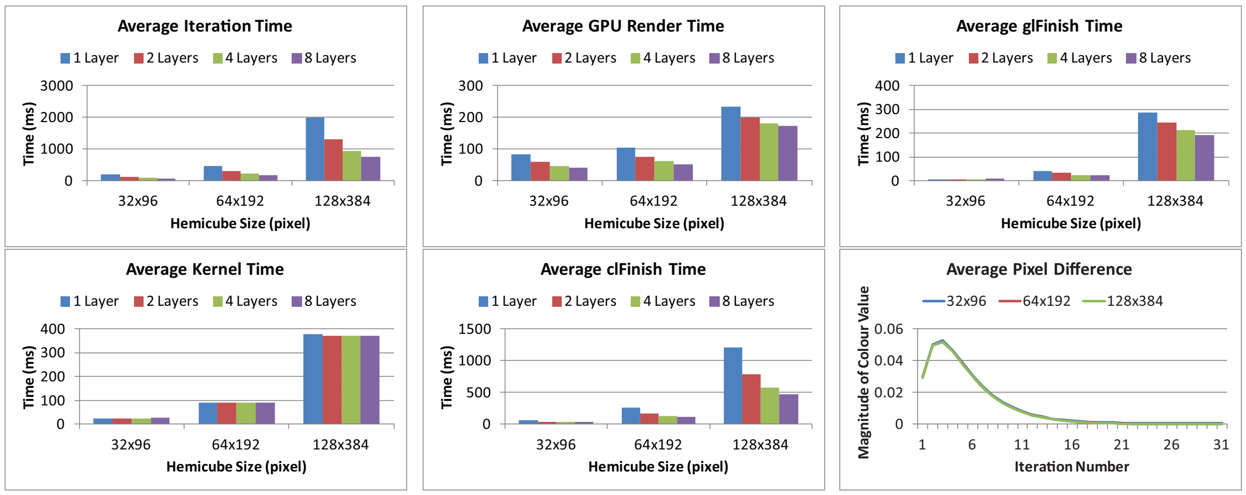


Fig. 4. Performance of different hemicube sizes and number of layers in a 1024x1024 texture array (average over 30 iterations).

TABLE I
COMPARISON OF CPU-BASED AND GPU-BASED IMPLEMENTATION.

Hardware	CPU	GPU			
Hemicube Size	128	128			
Num of Layers	1	1	2	4	8
Avg. Iteration Time (ms)	14770.22	1987.27	1308.25	951.82	767.95
Speed-up Factor	---	7.4	11.3	15.5	19.2
Hardware	CPU	GPU			
Hemicube Size	64	64			
Num of Layers	1	1	2	4	8
Avg. Iteration Time (ms)	3814.17	467.75	310.52	231.77	188.75
Speed-up Factor	---	8.2	12.3	16.5	20.2
Hardware	CPU	GPU			
Hemicube Size	32	32			
Num of Layers	1	1	2	4	8
Avg. Iteration Time (ms)	1108.60	196.75	124.92	98.66	86.80
Speed-up Factor	---	5.6	8.9	11.2	12.8

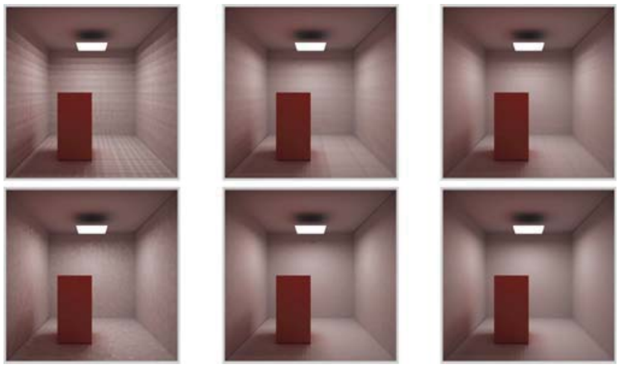


Fig. 5. Effects of hemicube resolution. 32x32x16 pixels (left), 64x64x32 pixels (center), 128x128x64 (right). Top row without rotation, bottom row with random rotation.

visible on higher hemicube resolutions, however, they increase the computational complexity. As shown by Fig.5, a compromise between computation cost and acceptable visual quality can be achieved using a relatively low resolution hemicube with a random rotation. In the sample scene, hemicube size of 64x64x32 pixels with a random rotation added to each patch's hemicube are acceptable. As the scene and/or lighting complexity increases, further optimisation will be necessary.

C. OpenCL Work Size

The number of work items and work groups along with how the GPU's global memory is accessed highly affect the efficiency of the computation. The number of work items needs to be a multiple of the size of the smallest work unit. 32 work items constitutes a unit of work and are executed at the same time. An occupancy level of higher than 25% is required to hide memory latency [13] and to ensure the GPU has enough work to do for all of its processing units. The sample scene has an occupancy level of 66.7% with 32x16x1 work items per work group for the first kernel, 32x1 for the second kernel and $hemicube_width$x1 for the last kernel. The global memory access for each unit of work also needs to be coalesced to obtain higher efficiency. This is done by ensuring that each work item accesses its designated memory location within the buffers and that each work item accesses sequential data from the global memory.

D. Dynamic Environment

The proposed method does not need to calculate or store the form factors between any pair of patches because a multiplier map is used. As position and/or orientation of any geometry is changed, its transformation can be applied to the *center* and *normal* data stored in the *patch list* to obtain the latest position and orientation from which to render the hemicube. In the case of real time performance, this will result in a smooth transition. As the lighting intensity changes, the estimated radiosity will adjust to the new lighting intensity values which will stabilise over time as more iterations are performed. This is similar to how iterations using the Gauss-Seidel method will converge towards the true solution as more iterations are performed regardless of the initial or current values.

VI. Limitation and Future Work

A. Banding Artifacts

Irradiance caching can be implemented to only sample high resolution hemicubes from key locations that are considered important (position of patches compared to occluders and reflectors) and interpolate between them. This will create a smoother appearance through interpolation and reduces the number of hemicubes to be rendered. Elias [11] suggested interpolation in texture space by rendering hemicubes for every 4^{th} pixel in the lightmap and interpolating between them. If the difference between the sampled values is higher than a certain threshold, a new hemicube is rendered and integrated.

B. Interpolation

The UV parameterisation process to create the UV coordinate set needs to be clean and precise. In the sample scene, the UV coordinate set is generated manually, with a UV island created for each of the 9 quads (similar to Fig.1). The UV coordinate of each vertex is forced to lie on a pixel boundary to ensure that the center of each pixel is enclosed by the quad, so that each patch is represented by a whole pixel. If the UV coordinate does not lie on a pixel boundary, artifacts might occur in the form of black patches in the rendered scene. In this case, the pixel is not rasterised but sampled in the rendered scene because no interpolation is used (using nearest sample).

If interpolation is used, boundaries of the UV islands will create issues as the edges of the quads in the rendered scene will have a darker appearance due to its interpolation with unused texels outside the boundary of the UV island (black by default). One solution to the interpolation and misaligned UV coordinates issues is to dilate the UV islands so that when interpolation is performed, the boundaries of the UV island will be interpolated with itself. This step should only be performed when displaying the result on the screen and not during the iterations to avoid adding energy to the scene. Another solution is to adjust the UV coordinates during the initial *patch list* creation using a geometry shader, each UV coordinate will move at most half a pixel to be aligned with the pixel boundaries.

C. Other Frameworks

The main limitation with the OpenCL 1.0 implementation is the need to call `glFinish()` and `clFinish()` to ensure buffers are synchronised. As seen by the impact the synchronisation overhead has on the overall iteration time, other computing frameworks should be used to compare their performances in this regard. These include OpenCL 1.1 and NVIDIA CUDA frameworks. A complete port using Microsoft DirectX graphics library along with its new DirectCompute capability could also be done.

VII. Conclusion

Radiosity is a highly parallelisable problem and as shown by the proposed method, performing the calculations on graphics hardware to exploit its computing power can improve convergence speed and performance. The improvement depends on the the numbers of cores and processor speed of the CPU for the CPU implementation and the number of shader cores and their speed for the GPU implementation. The proposed method's ability to support dynamic environments and interactive radiosity calculation means that it can be used in many applications such as architectural visualisations, simulations and eventually video game applications. As graphics hardware becomes more powerful and as their supporting frameworks become more mature, it can be envisioned that radiosity calculations will be performed in real time.

Acknowledgment

The authors would like to thank the graphics team at Sidhe: Alan Chambers, Robert Higgs, Kester Maddock, Robert Cannell and Ramon Steenson for their support and guidance. The authors would also like to thank MSI New Zealand for their support through the Fellowship Funding.

References

[1] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '84. ACM, 1984, pp. 213–222.

[2] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '86. ACM, 1986, pp. 143–150.

[3] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, "A progressive refinement approach to fast radiosity image generation," in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '88. ACM, 1988, pp. 75–84.

[4] A. Keller, "Instant radiosity," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56.

[5] M. F. Cohen and D. P. Greenberg, "The hemi-cube: a radiosity solution for complex environments," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '85. ACM, 1985, pp. 31–40.

[6] K. H. Nielsen and N. J. Christensen, "Fast texture-based form factor calculations for radiosity using graphics hardware," *J. Graph. Tools*, vol. 6, pp. 1–12, September 2002.

[7] G. Coombe, M. J. Harris, and A. Lastra, "Radiosity on graphics hardware," in *Proceedings of Graphics Interface 2004*, ser. GI '04. Canadian Human-Computer Communications Society, 2004, pp. 161–168.

[8] S. Martin and P. Einarsson, "A Real Time Radiosity Architecture for Video Games," in *SIGGRAPH '10: ACM SIGGRAPH 2010 Courses*, ser. SIGGRAPH '10. ACM, 2010.

[9] S. Martin, "Enlighten Research, past, present, future," ser. London Graphics Seminar 2011, 2011.

[10] I. Castaño. (2011) Hemicube rendering and integration. [Online]. Available: http://the-witness.net/news/2010/09/hemicube-rendering-and-integration/

[11] H. Elias. (2011) Radiosity. [Online]. Available: http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm

[12] J. Hensley, D. Gerstmann, and T. Harada, "Advanced opencl by example," in *SA '10: ACM SIGGRAPH ASIA 2010 Courses*, ser. SIGGRAPH '10. ACM, 2010.

[13] "OpenCL Best Practices Guide," White Paper, NVIDIA Corporation, February 2011.