Reconfigurable Designs for Radiosity

Paul Baker, Tim Todman, Henry Styles and Wayne Luk
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, England

Abstract

We develop reconfigurable designs to support radiosity, a computer graphics algorithm for producing highly realistic images of artificial scenes, but which is computationally expensive. We implement radiosity using stochastic ray-tracing, which affords both instruction-level and data parallelism. Our designs are parameterisable by bitwidth, allowing trade-offs between image quality and computation speed. We measure the speed of our designs for a Xilinx XC2V6000 device in the Celoxica RC2000 platform: at 53MHz it can run up to five times faster than a software implementation on an Athlon MP 2600+ processor at 2.1GHz. We estimate that retargeting our design for a Virtex-4 XCVSX55 device can result in over 160 times software speed, while a Spartan-3 XC3S5000 device can run more than 40 times faster than the software implementation.

1. Introduction

Real-time methods for calculating lighting in computer graphics use local illumination methods, where the colour of a given point on a surface depends only on the properties of that surface, and of a set of light sources. To produce more realistic images, we can use global illumination methods, where all surfaces in a graphics scene are considered as potential sources of illumination for all other surfaces. One specific technique for solving the global illumination problem is radiosity. This involves modelling the balance of energy between surfaces in a view-independent manner, hence accurately calculating the diffuse illumination of all objects in the scene. Unlike other graphics algorithms like ray-tracing, radiosity can model diffuse lighting effects, such as the bleeding of colours between nearby objects. Radiosity is an extremely computationally expensive process; it has mainly been used by film studios with large computers and much time to calculate lighting solutions. For example, to calculate the lighting for the seven-minute film *Bunny* by Blue Sky Productions took four weeks on a Compaq AlphaServer RenderPlex system using a total of 164 processors [1]. Although the radiosity technique is computationally expensive, the majority of calculations are involved in repetitively answering a simple question: "does this line segment intersect this triangle?". This paper investigates using reconfigurable hardware to accelerate the calculation of a radiosity solution.

The achievements covered by this paper include: (a) a hardware architecture for accelerating calculations involving intersections of triangles and line segments (section 3), (b) optimisations including rearranging computation ordering to reduce memory bandwidth, and concurrent execution of hardware and software elements (section 4), (c) implementation and evaluation of the proposed approach, showing that a single FPGA with clock speed almost 40 times slower than a microprocessor can outperform the latter by up to 5 times (section 5).

Although our design does not take advantage of run-time reconfiguration, and thus could be implemented as an Application Specific Integrated Circuit, implementing it on reconfigurable hardware has two key benefits: (1) our generic description, parameterised by bitwidth, can be used to create several designs with different speed and fidelity trade-offs, allowing users to choose their preferred solution to suit particular domains; (2) we have studied FPGA-specific optimisations, such as the use of Virtex II block multipliers.

2. Background and related work

The radiosity method [2, 3] models the balance of energy between surfaces in a view-independent manner, hence accurately calculating the diffuse illumination of all objects in the scene. These objects are generally made up of polygonal faces. These faces are further split up into *patches*, which are the individual units considered by the radiosity calculations. The radiosity B of a surface is the energy leaving the surface per unit area [4]. This is the sum of the emitted energy E and the reflected energy. For a small area dA, we have:

$$BdA = EdA + RI$$

where R is a constant, the reflectivity of a surface, and I is the incident energy. The incident energy I_i at each patch i is collected from all other patches:

$$I_i = \sum B_j F_{ij}$$

where F_{ij} is a constant which links patch i and patch j called the *form factor*. This gives:

$$B_i = E_i + R_i \sum_j B_j F_{ij}$$

The aim is to solve this for all B_i, that is, to calculate the radiosity of every patch in the scene. The form factor between patches i and j is given by:

$$F_{ij} = \frac{V_{ij} cos(\phi_i) cos(\phi_j) Aj}{\pi r^2}$$

where ϕ_i and ϕ_j are the angles between the line connecting the centres of the two patches and the normal of each patch, A_j is the area of patch j, and r is the distance between the centres of the two patches. This is illustrated in figure 1. V_{ij} is the visibility: how much of patch j is visible from patch i. Assuming the patches are small, V_{ij} can be simplified by considering only whether there is a clear line of sight between the centres of the two patches:

$$V_{ij} = \begin{cases} 1 & \text{if a clear line of sight exists} \\ 0 & \text{otherwise} \end{cases}$$

The above problem can be reformulated as a matrix equation:

$$\begin{bmatrix} 1 & -R_1F_{12} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1 & \dots & -R_2F_{2n} \\ -R_3F_{31} & -R_3F_{32} & \dots & -R_3F_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ -R_nF_{n1} & -R_nF_{n2} & \dots & 1 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ \vdots \\ B_n \end{bmatrix} \begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ \vdots \\ E_n \end{bmatrix}$$

This can be solved using the Gauss-Seidel iterative method:

$$\begin{array}{ll} \text{Initially :} & B_i^0 = E_i \\ k^{\text{th}}\text{iteration :} & B_i^k = E_i + R_i \sum B_j^{k-1} F_{ij} \end{array}$$

This method allows a partial solution to be rendered after each step, so that the results of the calculation can be previewed. The method of progressive refinement [6] is a slight alteration to the above method, which can considerably reduce the number of iterative steps required. Evaluation of a single radiosity value using the Gauss-Seidel method is a process of gathering. For the patch whose radiosity value we wish to compute, all of the other patches are examined,

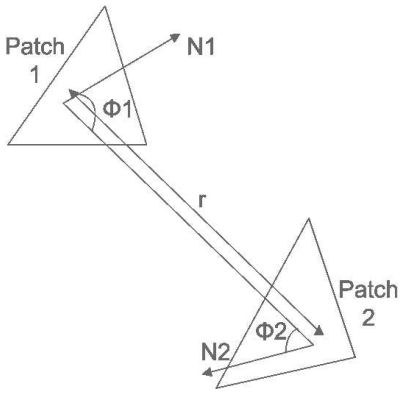

Figure 1. The angles and distance used in form factor calculations

and the amount of energy they pass to the first patch is calculated. This is essentially solving the matrix equation by considering one row at a time.

Another way to solve the matrix equation is to consider a column at a time. If in a given iteration B_i changes by ΔB_i, the radiosity of each other patch will need to be updated by:

$$\Delta B_j = R_j F_{ji} \Delta B_i$$

giving:

$$B_j^k = B_j^{k-1} + R_j F_{ji} \Delta B_i$$

Rather than the gathering of the Gauss-Seidel method, this is a process of shooting. For each iteration, a patch is chosen and its energy is distributed to all of the other patches in the scene. This allows us to reduce the number of steps by always choosing to distribute from the patch with the largest amount of energy. The computation ends when a simple heuristic is satisfied: if the maximum unshot energy (that yet to be distibuted) is less than a certain fraction of the maximum initial energy, then the computation has converged. For example, if the maximum unshot energy is less than 0.5% of the maximum initial energy, 99.5% convergence has been achieved.

Calculating the visibility term between two patches is the most computationally expensive part of the radiosity algorithm. Given patch i and patch j, a line segment from the centre of patch i to the centre of patch j can easily be constructed. Using the above approximation, the visibility term is a simple test: does this line segment intersect any other triangle? If it does, we consider the patches to be non-visible from each other, and thus set V_{ij} to be zero. If there is no intersection, the visibility term is set to one.

Although the intersection calculations are simple, the overall complexity is prohibitive. For each step in the algorithm, energy needs to be transferred from a single source

patch to every face. For m faces in the scene, and n patches, for a single step each ray needs to be tested for intersection against every face, leading to $O(mn)$ intersection calculations. Given that the number of steps required to reach a certain level of convergence is linear in the number of patches, the radiosity algorithm overall needs $O(mn^2)$ such calculations. To reduce the number of intersection calculations necessary, spatial subdivision methods can be used; these typically enclose several patches within a bounding volume; if rays do not intersect the bounding volume, they cannot intersect the patches contained within.

We now consider related work. Styles and Luk [12] study the feasibility of implementing radiosity on reconfigurable hardware. They show that a throughput of 5.4 million ray-triangle intersections is possible using a Xilinx XCV2000E-6 FPGA chip, and estimated 22.4 million intersections per second should be achievable with an XC2V8000-4 device. Ko and Ng [13] develop an architecture that reconfigures between visibility calculations and radiosity distribution, and attain roughly four times speedup over software. Their design moves configurations to and from the hardware, rather than just data, as we do. It is possible that we could use this technique with our design, if host to hardware bus speed becomes a limiting factor.

There are several examples of using reconfigurable hardware to accelerate ray-tracing, a related rendering algorithm which relies heavily on ray-object intersection calculations. Fender and Rose [14] use two Virtex 2000E FPGAs for their ray-tracing architecture, and achieve a peak 23 times speed up over a software implementation for a scene with 512000 triangles, and an average speed up of 9 times over a range of scenes. The SaarCOR project [15] is another hardware architecture for ray-tracing, which uses the same RC2000 board as is used in this paper. The peak speedup achieved by the SaarCOR team is 4.7 times, with an average improvement of 3.7 times. Ray-tracing requires not only intersection tests to be computed, but also lighting coefficients. The SaarCOR design thus uses a novel method which can calculate both an intersection and a ray-normal dot product in a single operation. A more efficient ray-triangle intersection method should be used if intersection calculations are all that is required, as in this paper.

Some work has also been done on using programmable graphics hardware to accelerate radiosity calculations. This hardware is not reconfigurable, rather it is programmable in the same sense as a general purpose CPU. However, the graphics hardware still runs independently of the host CPU. Coombe, Harris and Lastra [16] use an Nvidia Geforce FX5900 graphics card hosted on a Pentium 4 running at 1.8 GHz and achieve an average of 263 lighting steps per second. In contrast, the design we develop is capable of producing 271 light steps per second; details can be found in Section 5.

3. Mapping to hardware

Since the visibility calculations are by far the most computationally demanding part of a radiosity solver, we choose to implement these on the reconfigurable hardware, whilst the rest of the steps of the algorithm are carried out on the host PC. In this section, we discuss the issues involved in developing a hardware architecture to perform these calculations:

- Evaluating a ray-triangle intersection algorithm to use, in section 3.1
- Increasing the parallelism of an architecture for this algorithm, in section 3.2
- Using fixed-point arithmetic, in section 3.3
- Limiting the widths of fixed-point intermediate variables, in section 3.4

3.1. Ray-triangle intersection

There are several ray-triangle intersection algorithms which could have been used for this paper. We use Möller and Trumbore's algorithm [17] as it does not require the plane equation of the triangle to be stored or computed. This saves memory bandwidth as the plane equations need not be passed to the reconfigurable hardware.

The algorithm proceeds as shown in figure 2, where "$\wedge$" represents a vector product and "$\cdot$" represents a scalar product. The inputs to the algorithm are the five three-dimensional vectors: start (Line segment start position), direction(Line segment end position), v0 (Triangle vertex 0), edge1 (Triangle edge 1) and edge2 (Triangle edge 2).

To calculate values for u, v and t, a division by det is needed, which is expensive in hardware resources. Note that the only operation performed on u, v and t after their computation is a comparison: does the value lie in [0, 1]? The division can thus be removed by altering the comparison. Rather than testing whether $0 < \frac{u}{det} < 1$ a calculation which gives the same results is $0 < \frac{u}{sign(det)} < |det|$. The second calculation avoids the division, consuming fewer hardware resources. The sign of det can easily be taken since det is a two's complement number, and hence its sign will be given simply by the value of its most significant bit. The modulus of det can be calculated by negating det to give $-det$, and then using the sign bit to select between det and $-det$.

3.2. Increasing parallelism

The data dependency graph in figure 3 reveals instruction-level parallelism. The temporary variables temp1, temp2, temp3, det, u, v and t depend on each other and on the input values. In this graph, an edge from one

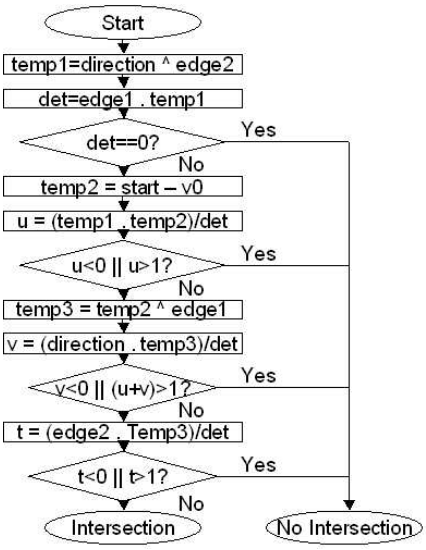

Figure 2. Flowchart of Möller-Trumbore algorithm

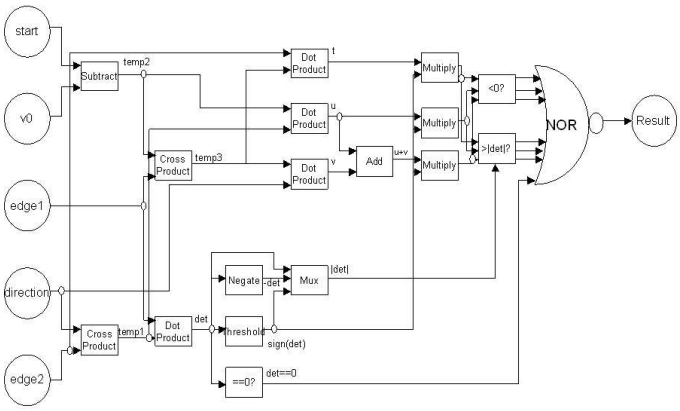

Figure 4. Architecture for Möller-Trumbore algorithm. Most blocks are pipelined in our implementation in section 5.1.

variable to another means that the second variable depends on the first. We can split the algorithm into three stages of calculation, computing several variables in parallel at each stage: Stage 1 (temp1, temp2), Stage 2(temp3, det) and Stage 3 (u, v, t).

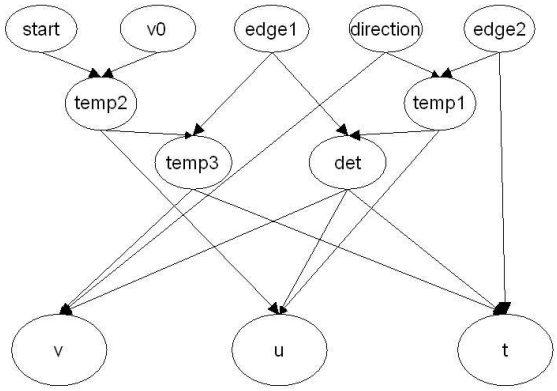

Figure 3. Data dependency graph for Möller-Trumbore algorithm

Also, the operations required once det has been computed, such as calculating $|det|$, can be performed in parallel with the calculation of u, v and t, since the value of $|det|$ is not required until the necessary vector operations to compute u, v and t have been completed. This leads to the architecture given in figure 4.

The individual operations which calculate these quantities can also take advantage of parallelism. Calculating a vector product requires six multiplications and three subtractions to be performed. The multiplications can be performed in parallel, as can the subtractions, which rely on the multiplication results. The scalar product requires three multiplications, all of which can be parallelised, but the results then need to be added together. Thus two serial additions are required after the multiplications are complete.

3.3. Fixed-point arithmetic

In our software implementation of the radiosity solver, all calculations are done using floating-point arithmetic. To maximise the speed and minimise area, the hardware implementation uses fixed-point arithmetic. To simplify the conversion from floating-point to fixed-point, all scenes are scaled such that the X, Y and Z coordinates of their vertices lie in the range [0, 1]. Thus, the absolute positions which need to be passed to hardware, for example the positions of vertices, can be converted into an unsigned fixed-point format with no bits before the binary point, and N bits afterwards. Differences between two points, for example the direction vector of a line segment, can be expressed in signed fixed-point with a single bit before the binary point.

Fixed-point arithmetic has a much smaller dynamic range than floating-point arithmetic in a similar bitwidth. However, by constraining the scene within the range [0, 1], the dynamic range of values within the Möller-Trumbore algorithm is similarly constrained.

Our fixed-point architecture is parameterised by the number of binary places used. This allows users to manually trade speed for fidelity: more binary places give greater fidelity at the expense of speed; section 5.1 shows the trade-off between speed and number of binary places.

3.4. Reducing intermediate variable widths

Calculating the result of an intersection using Möller and Trumbore's algorithm requires the calculation of two vector cross products and four dot products. Including the final multiplication of u, v and t by the sign of the determinant, a total of 27 multiplications, the algorithm requires 27 multiplications in total. Multiplication of an A.B fixed-point number (that is, one with A integer bits and B fractional bits) by a C.D fixed-point number produces a result with A + C bits before the binary point and B + D bits after, resulting in an (A+C).(B+D) fixed-point number. Since the result of one multiplication is often an argument to another (see figure 4), significant fan-out can occur, leading to quantities with very large fractional parts. If all calculations are performed to full precision and numbers are left unadjusted, the wide logic required to perform arithmetic on the numbers causes a substantial reduction in maximum clock speed and an increase in the area required for the circuit.

In order to prevent this from causing a problem, all results from multipliers are truncated so that they have the same number of bits after the binary point as the original inputs to the circuit. Thus, the number of fractional bits in the values throughout the algorithm are the same.

The number of integer bits does not grow as quickly as the number of fractional bits, since there are many fewer integer bits to begin with. For example, if two inputs to the pipeline are 1.15 fixed-point numbers, with one integer bit and 15 fractional bits, multiplying them together will result in a 2.30 number. Multiplying this by another, similar product leads to a 4.60 fixed-point number. Clearly the number of fractional bits is becoming very large, whereas the number of integer bits has only increased by 3. Furthermore, none of the integer bits can be discarded, since doing so would lead to the calculations overflowing the integer bits. Thus, only the fractional bits are truncated, so in this example, the number would be reduced to 4.15 format.

4. Optimising hardware

The architecture developed thus far has several opportunities for optimisation. In this section we discuss some of the tweaks applied to both the hardware and software sides of the radiosity solver. Specifically, these are:

- Reducing memory bandwidth by rearranging the order of computation, in section 4.1
- Exploiting parallel computation by creating two copies of the intersection pipeline running in parallel, in section 4.2
- Using the time during which the FPGA is calculating intersection results to perform useful calculations on the host PC, in section 4.3
- Increasing the upper bound on the number of faces and patches in scenes for which the hardware can be used, in section 4.4

4.1. Reducing memory bandwidth

The pipeline delivers a result every cycle. As inputs, the pipeline requires 15 fixed point numbers, (five 5 three-dimensional vectors): line segment start position, line segment end position, triangle vertex 0, triangle edge 1 and triangle edge 2.

For a single source patch, the line segment start position is fixed. For each destination patch there will be a fixed line segment end position. Every face of the scene needs to be tested for intersection with this line segment. To deliver one result per cycle therefore, 9 fixed point numbers need to be input to the pipeline per cycle.

We can significantly reduce memory bandwidth by rearranging the order of computation. Rather than fixing a line segment and testing for intersection against all of the faces, we fix a face, and test against all of the line segments from the source patch to the destination patches. Since all of the line segments will begin at the centre of the source patch, only the line segment end position need be updated each cycle. Thus only 3 fixed point numbers need to be input to the pipeline per cycle.

If 9 fixed point numbers were required per cycle, each of at least 16-bit precision, these would need to be read from five 32-bit wide memory banks. Assuming six memory banks are available, in order to fit all of the data required into those banks, all line segment data would have to be placed in the final bank. Once a line segment has been considered, its data could be overwritten in the sixth memory bank by the result of its intersection tests.

Reducing the requirement to 3 fixed-point numbers per cycle simplifies the design of the memory system. One of the six banks of memory can be used to hold the face data, and one bank each can be given to the X, Y and Z coordinates of the line segment end positions. This way, the line segment coordinates can be up to 32 bits wide, and all three coordinates can be read in parallel in a single cycle. Once the result has been computed, this can be extended to 32 bits and placed in a fifth memory bank. This memory organisation and the way it is used to stream data through the pipeline is shown in figure 5.

If there are six or more memory banks then we use the following strategy. Our basic architecture requires five memory banks, but some platforms have eight banks or more. Each ray intersected in parallel requires three banks, so N rays require $3N + 2$ banks. For platforms with six or seven banks, we can modify this strategy to place the rays in one or two banks. Those banks then feed a pipeline which is deliberately run slower than the main pipeline, be-

cause it is only fed by one or two banks, running at one third (for six banks) or two thirds (seven banks) of the speed of main pipeline. The speedup over five banks is then $4/3$ (six banks) or $7/3$ (seven banks).

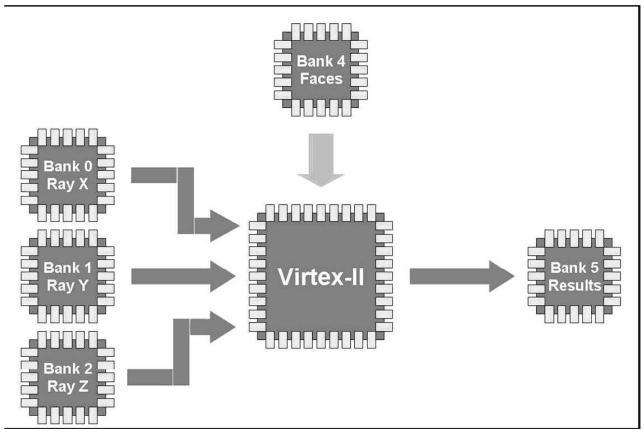

Figure 5. Example assignment of memory banks

4.2. Parallel computation

As shown in the previous subsection, it is possible to reduce the data required by the pipeline to only three fixed-point numbers per cycle. Another benefit of this reduction is that if the fixed-point coordinates are restricted to being 16-bits wide, each 32-bit word in the memory bank holding line segment X coordinates can hold two numbers. Thus, two X coordinates can be read in per cycle. Similarly, two Y and Z coordinates can be read per cycle. Two results can easily be written in a single cycle, since each result is only a single bit. If two copies of the pipeline can be fitted onto the reconfigurable device in parallel, it is possible to increase throughput to two results per cycle. This is a kind of data parallelism: the same instruction (the intersection algorithm) is simultaneously applied to several data items (the rays).

To take advantage of the presence of two copies of the pipeline, the wrapper is altered slightly. Whenever the face data or ray start data is read from memory, this is passed to both copies of the pipeline in parallel. When ray data is read, each individual 32-bit word contains a coordinate for each of two rays, so this is split and one coordinate is sent to either pipeline. Thus the two copies of the pipeline allow the calculations to occur with two rays at once, but the same triangle is used as a potential shadow caster in each copy.

One slight complication with this method is that it is no longer possible to compute the logical-or of the result of the intersection being carried out with the result already in memory by simply ignoring the result if it is not blocked. This is because each memory word will now hold two results, and the logical-or of the two results needs to be taken independently. Thus it is necessary to read back the previous result from memory and logical-or the results together explicitly.

4.3. Concurrent execution: software and hardware

For a single batch of rays and triangles, the host program uploads data to the FPGA board and then informs the chip that the data is present. The FPGA will then begin processing the data and will signal an event, via an interrupt, once calculations are complete. The algorithm described in the previous section has the host PC simply waiting for this event immediately it has told the card to begin processing.

Although the FPGA is responsible for the intersection calculations, the host PC must still calculate the form factors between the source and each destination patch. This will be modulated by the intersection test results once they are available. If an array of a large enough size to hold one floating-point number per patch is created on the host PC, the form factors can all be computed in parallel to the intersection tests occurring on the FPGA board. These can be stored in the array and read back when the energy transfer is actually calculated. Some pseudocode demonstrating this is shown below:

```
UploadData(rays, triangles);
WriteFPGA(FPGA_START_CALCULATION_ADDRESS, 1);
//Calculate form factors whilst the hardware
// calculates visibility terms
float formFactors[numPatches];
for(int i=0; i<numPatches; ++i)
   formFactors[i]=(cos(phi1)*cos(phi1)
    * shootingPatch.area)/(PI*r*r);
//Wait until calculations complete
WaitForEvent(interruptEvent);
//Continue to distribute energy
//...
```

4.4. Raising upper bound on number of faces and patches

One limitation of the implementation described in the previous section is that, with limited memory banks, all the patches from the scene must fit in one bank. For example, 4MB banks (common on many reconfigurable platforms) limit us to scenes with fewer than 116,000 ($\simeq$ 4MB divided by 9) faces and 2^{20} patches. Very large scenes may contain more faces than this limit, and scenes where very fine detail is required may include more patches than can be processed.

To cope with an increased number of faces, multiple uses of the FPGA hardware can be made in a single lighting step. The first 116,000 faces can be considered as usual, and the

results read back. The calculations can then be done again, using the next set of faces. Once the results for each set of faces are known, a simple operation will combine them to give overall results; a ray is blocked overall if it is blocked in any of the result sets. If an increased number of patches is required, again multiple uses of the hardware can be used to calculate full results. Since the rays are streamed through the hardware independently, all that is required is to calculate intersections using the first set of rays, then upload the second set of rays and calculate results for them.

These two techniques can be combined to compute radiosity solutions for scenes with arbitrary numbers of faces and patches. The first step is to split the rays into batches of up to 2^{20} each and upload the first batch. Then, for each set of up to 116,000 faces, calculate the intersections with the rays, combining the results to give overall intersection values for each of the first 2^{20} rays. The calculations can then be repeated for each further batch of rays.

By doing the calculation in this order, rather than uploading a batch of faces at a time and streaming through all of the rays, the amount of traffic over the PCI bus is reduced since the face data for a single calculation run is only 4MB, whereas the ray data is 12MB. Also, after the first batch of rays have been tested against all of the faces, the results can be used to transfer energy and then discarded, so that enough memory to hold the results of 2^{20} intersection tests is required on the host PC, rather than requiring the temporary storage of results for all of the rays.

5. Implementation and evaluation

5.1. Implementation

The hardware implementing the Möller-Trumbore algorithm, shown in figure 4, is generated using Xilinx System Generator [9] and driven by a wrapper written in Celoxica's Handel-C [8]. The pipeline consists of 18 addition/subtraction blocks and 27 multipliers, as well as some logic elements. It has a total latency of 37 cycles and a throughput of one result per cycle.

5.2. Varying pipeline precision

The pipeline for ray-triangle intersection takes fixed-point numbers with zero or one integer bit and a fixed number of fractional bits as inputs. The number of fractional bits can be varied to trade off speed and size versus accuracy. The following table shows the results, from place and route reports, of instantiating a single copy of the pipeline, without any support circuitry, on the XC2V6000 FPGA chip.

Fractional Bits	Max. Clock Rate / MHz	No. of Slices (% of 33792)
7	219.5	1951 (5.77%)
15	185.5	5471 (16.2%)
23	132.3	10615 (31.4%)
31	59.6	16889 (50.0%)

These results show that as the precision of the calculations are increased, the size of the implementation increases and the maximum speed decreases. This corresponds with what would be expected; as the width of the operators in the circuit increases, their size increases, and so does the length of the critical path within them, thus causing the maximum clock speed to decrease. Figure 6 shows the maximum clock speed and number of slices versus the number of fractional bits.

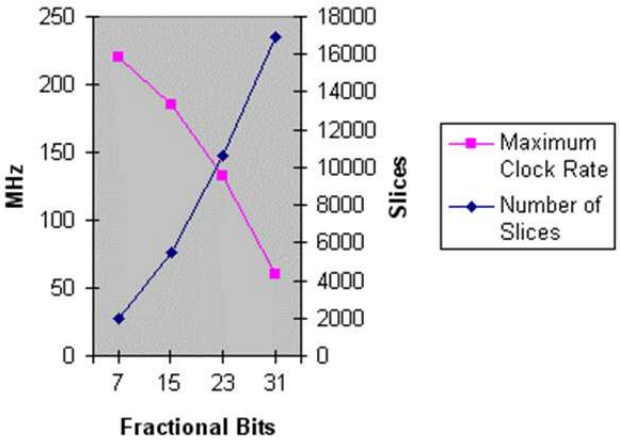

Figure 6. Graph of design speed versus fractional bitwidth

5.3. Using dedicated multipliers

Allowing the multiplication blocks in the pipeline circuit to take advantage of the 18 by 18 bit hard-wired multipliers provided in the Virtex-II chip changes both the maximum speed and the size of the design. Unfortunately, it is not possible to synthesize the 23 and 31 fractional bit precision pipelines if all multiplications in the circuit use the block multipliers. For the wider designs, a compromise is necessary; block multipliers can be used for some of the multiplications required, but not for others. Table 1 shows the size and speed of the 7 and 15 fractional bit precision pipelines when block multipliers are used. The last column gives the proportion of slices used compared to the previous table (when the use of block multipliers was disabled).

The block multipliers allow for a marked decrease in the number of slices used, since the multiplication blocks need

Fractional Bits	Max. Clock Rate / MHz	No. of Slices (% of 33792)	No. of Block Multipliers (% of 144)	Slices Used Compared to Non-Multiplier version
7	200.7	1128 (3.34%)	27 (18.8%)	57.8%
15	176.9	1976 (5.85%)	30 (20.8%)	36.1%

Table 1. Size and speed of pipeline when using dedicated block multipliers

No. of faces	No. of patches	Lighting steps required	Time taken / seconds				Overall speedup
			software	single hardware pipeline	2 parallel hardware pipelines	concurrent hardware software	
70	280	385	0.9	0.9	0.8	0.8	1.1
70	1120	1488	8.5	8.2	6.4	5.8	1.5
70	4480	5889	119.7	114.0	87.3	78.4	1.5
280	280	385	2.0	1.6	1.3	1.2	1.7
280	1120	1488	25.3	15.5	10.9	9.4	2.7
280	4480	5889	380.2	220.7	143.7	123.0	3.1
1120	1120	1488	93.8	45.6	27.7	23.1	4.1
1120	4480	5889	1459.8	651.7	370.0	298.5	4.9

Table 2. Comparing speedup from different implementations

no longer be synthesized. However, using the block multipliers does slightly reduce the maximum clock speed of the design. This is probably due to the fact that the multipliers are in fixed positions on the FPGA chip, and the remaining components need to be placed around the position of the multipliers. On the version of the pipeline with synthesized multipliers, the placement of components need not be constrained due to having to work around multiplier positioning, so can be more optimal.

5.4. Solutions for real scenes

Instantiating both the 15-bit fractional precision pipeline which uses block multipliers and the supporting circuitry produced by the Handel-C Wrapper on the Virtex-II chip gives the following speed and size results:

Max. clock rate / MHz	No. slices	No. Block Multipliers
53.833	3339 (9.8%)	30 (20.8%)

We use the combination to calculate radiosity solutions for several different incarnations of the Cornell Box scene [18], a standard radiosity test scene, based upon a real model. Physical data including emission and reflectance spectra have been measured from the real model and are used by the radiosity solver.

The results can be compared against real photographs of the box available from Cornell. Figure 7 shows the Cornell box scene; unfortunately much of the effect is lost due to the greyscale image. The side walls are red and green – only the radiosity algorithm can model the way the wall colour bleeds onto the white boxes. This effect cannot be realised using either local illumination methods, nor by using raytracing without radiosity.

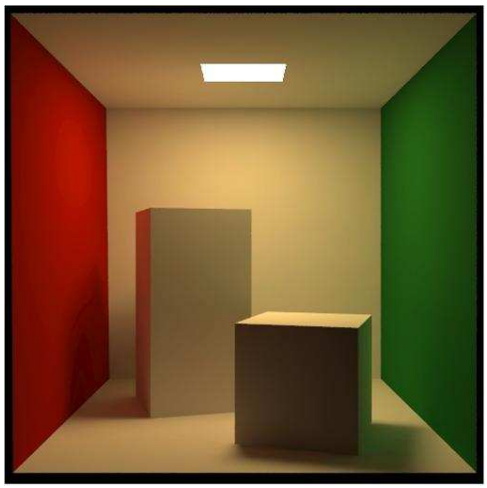

Figure 7. Cornell box test scene

The model of the box which we created for testing the radiosity solver is based on this physical data, and begins

with 70 faces each also constituting a single patch. The faces can be split uniformly to produce scenes with higher face counts for testing, and then these faces can be split further into patches. Table 2 shows the time taken by both a software implementation and the hardware radiosity solver to calculate the solution to 99.5% convergence for several different face and patch counts. All results were obtained from a dual processor Athlon MP 2600+ machine running at 2133MHz with 2GB of RAM and with the FPGA at its maximum speed of 53MHz.

The table shows that for all face and patch counts tested, the hardware implementation performs at least as well as the software implementation, and in many cases performs substantially better. On average, the hardware implementation provides a 50% speed-up over the software implementation, with a 124% speed-up in one case. In fact, as the scenes get larger, the amount by which the hardware implementation is faster than the software only solver increases, which implies that for even larger scenes, the amount by which the hardware solution is faster than a software-only approach will increase.

5.5. Multiple pipeline instances in parallel

The previous section shows how multiple pipeline instances could be run in parallel, producing multiple results per cycle. The following table gives size and speed results for two parallel instances using the RC2000 board (Virtex XC2V6000):

Max. clock rate / MHz	No. slices (%)	No. Block Multipliers
52.5	5562 (16.5%)	60 (41.7%)

The maximum clock rate is very close to the single-pipeline case, because the two pipelines are identical and independent.

5.6. Performance estimations

We now estimate potential performance when using recent reconfigurable devices, and when some constraints of existing platforms are removed. These estimations show the potential of radiosity on reconfigurable devices. We investigate (a) maximum performance when limited by memory bandwidth, and (b) maximum performance when limited by the resources of current reconfigurable devices.

(a) *Limited by memory bandwidth*. Many current reconfigurable platforms use several small static memory banks; the RC2000 that we use has six 4MB banks. Assuming a peak memory clock of 100MHz, the total available bandwidth is $0.1\text{GHz} \times 6\text{banks} \times 4\text{bytes}$ = 2.4GB per second. Each ray intersection reads three 16-bit words, and writes a 1-bit result: 6.125 bytes (we ignore reading new faces; these reads will be infrequent compared to intersection reads in large scenes). Limited by the RC2000 bandwidth, we can achieve: $2.4\text{GB}/\text{sec} \div 6.125 = 392 \times 10^6$ intersections / sec. This is roughly 30 times our measured peak software intersection rate.

(b) *Limited by current hardware device resources*. Recent devices can fit more instances of the pipeline. The Xilinx Virtex 4 XCVSX55 device (24,576 slices and 512 block multipliers) could accommodate 12 instances of the 15-bit pipeline. Assuming that this configuration could achieve the same speed in table 1 (176.9 MHz), this configuration would support 12 pipes $\times$ 176.9 MHz = 2122 millon ray intersections per second: 164 times speedup over our software implementation. This would require 6.125 bytes per cycle $\times$ 12 pipes = 73.5 bytes per cycle, requiring 73.5 $\times$ 176.9 = 12.9 GB / second bandwidth.

Is this feasible? Modern graphics cards have comparable bandwidth: the ATI Radeon X600Xt use DDR memory to achieve 12 GB / second memory bandwidth.

Similarly, we estimate that the Spartan 3 XC3S5000 device, with 33,280 slices and 104 block multipliers could fit three 15-bit pipelines. At 176.9MHz, this would require 3 pipes $\times$ 176.9MHz $\times$ 6.125 bytes per intersection = 531 million intersections per second, 41 times speedup over software. This requires 6.125 bytes per intersection $\times$ 3 pipes = 18.375 bytes per cycle or 3.23 GB / second bandwidth at 176.9MHz.

5.7. Summary

In this section, we first give the speed and size results for the ray-triangle intersection pipeline at different precisions, both with and without the use of block multipliers. We then present the speeds achieved by four different implementations on scenes of ranging complexity. These results are summarised in table 2.

Comparing the version of the hardware which calculates two intersections in parallel per cycle controlled by the enhanced driver, we obtain a significant overall improvement in speed using the techniques here. The average improvement over the 8 scenes tested is 2.6 times, and for the largest scene tested, the speedup is just under five times.

The maximum theoretical number of intersections per second possible using the implementation we have developed for the XC2V6000 chip is 104 million (2 parallel pipelines at 52 MHz). Compared to the feasibility study [12], which claims a maximum of 22.4 million intersections per second on an XC2V8000 FPGA, our implementation shows a 4.6 times improvement in throughput, whilst using a smaller FPGA.

The implementation of radiosity using graphics hardware presented by Coombe, Harris and Lastra [15] achieves an average of 263 lighting steps per second. Our implementation achieves an average of 271 steps per second for the

same scene. The two approaches are very different, yet give almost identical performance. However, the graphics hardware used runs at a clock speed of 400MHz, almost eight times that required by our implementation.

6. Conclusion

This paper develops reconfigurable designs to support the radiosity algorithm, with three contributions: (a) a 37-stage fixed point hardware pipeline with user-defined bitwidth for the Moller-Trumbore ray-triangle intersection, (b) optimsations such as re-arranging order of computation and introducing multiple pipelines running in parallel, (c) evaluation of the proposed approach, showing speed improvement of up to 5 times over a software implementation at almost 40 times the clock speed.

Current and future work includes the following. First, floating-point implementation. We carefully normalised our scene and arranged our fixed-point implementation so that it would not overflow, whilst maintaining a constant fractional bitwidth in the ray-triangle intersection algorithm. A floating-point version would yield much higher dynamic range, reducing the need to normalise. It could also reduce the rounding errors inevitable from truncation. Although floating-point arithmetic takes much more hardware resources than fixed-point, this could be offset by being able to use a narrower total bitwidth, saving on bandwidth.

Second, other subdivision methods. Our software implementation used a regular subdivision of the scene into patches. More sophisticated versions of the radiosity method adaptively subdivide the scene according to how the illumination changes over the surfaces, so more rapidly changing areas are accorded more subdivisions, and subdivisions align with discontinuities such as shadow boundaries.

Third, multiple FPGA cards. Our evaluation shows that multiple parallel pipelines perform well. The radiosity method is sufficiently parallel that several FPGA cards could usefully work simultaneously on the same scene. This would require careful planning to avoid one card waiting for more inputs whilst another accesses the bus.

References

[1] H. Madden, *Nothing But Blue Sky*, `http://www.newenglandfilm.com/news/archives/99october/bunny.htm`, October 1999

[2] C.M. Goral, K.E. Torrance, D.P. Greenberg and B. Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces", *Siggraph 84*, 213-222, 1984

[3] T. Nishita and E. Nakamae, "Continuous Tone Representation of Three- Dimensional Objects Taking Account of Shadows and Interreflection", *Siggraph 85*, 23-30, 1985

[4] D. Gillies, Computer Graphics Lecture Notes - Radiosity, 2004

[5] M.F. Cohen, S.E. Chen, J.R. Wallace and D.P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation", *Siggraph 88*, 75-84, 1988

[6] M.F. Cohen and J.R. Wallace, *Radiosity and Realistic Image Synthesis*, AP Professional, 1993

[7] F.X. Sillion and C. Puech, *Radiosity and Global Illumination*, Morgan Kaufmann, 1994

[8] Celoxica, DK Design Suite, `http://www.celoxica.com/products/tools/dk.asp`, 2004

[9] Xilinx System Generator, `http://www.xilinx.com/products/software/sysgen/features.htm`, 2004

[10] Xilinx, ISE, `http://www.xilinx.com/products/designresources/designtool/index.htm`, 2004

[11] Celoxica, RC2000 Development Board, `http://www.celoxica.com/products/boards/rc2000.asp`, 2004

[12] H. Styles and W. Luk, "Accelerating Radiosity Calculations Using Reconfigurable Platforms", *Proc. FCCM 02*, 2002

[13] J. Ko and K. Ng, "Reconfigurable Implementation of Radiosity Distribution Computation", *Proc. FPT 02*, 2002

[14] J. Fender and J. Rose, "A High-Speed Ray Tracing Engine Built on a Field-Programmable System", *Proc. IEEE Int. Conf. On Field-Programmable Technology*, 188-195, December 2003

[15] J. Schmittler, S. Woop, D. Wagner, W.J. Paul and P. Slusallek, "Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip", *Graphics Hardware 2004*, 2004

[16] G. Coombe, M.J. Harris and A. Lastra, "Radiosity on Graphics Hardware", *Graphics Interface 2004*, 2004

[17] T. Moller and B. Trumbore, "Fast, Minimum Storage Ray-Triangle Intersection", *Journal of graphics tools*, 2(1):21-28, 1997

[18] Cornell Box Data, Cornell University Program of Computer Graphics, `http://www.graphics.cornell.edu/online/box/data.html`, 1998