

Directed Rendering

(c) Nicholas Blachford 31st January 2007

This document describes techniques designed to accelerate the production of high quality graphics at high frame rates using the image generation techniques called photon mapping and raytracing. The techniques are specific to the use of one or more Cell BE processor/s.

In order to achieve the acceleration I propose the use of “directed rendering”, this makes the random, unpredictable nature of photon mapping & raytracing into a predictable problem more suited to the Cell BE processor. Additional techniques similar to existing techniques applied in rasterization are discussed.

A system using similar techniques for audio is also described.

This document was written as part of a proposal for an 8th generation entertainment system (i.e. PS4) so mainly refers to games. However the techniques should work on the existing Cell BE and have the potential to give a substantial performance increase in any graphics related areas.

Index

Introduction	3
Making Photon Mapping a Cell Friendly Problem	4
Directed Rendering	5
<i>Local Map Generation</i>	5
<i>Saving Memory With Geometry</i>	5
<i>Photon Mapping</i>	5
<i>Raytracing</i>	6
<i>Updating The local Map</i>	6
Issues	7
Rendering Distant Objects	8
Antialiasing	9
Use of Objects Instead of Triangles	10
Texture Processing	11
Raytracing Audio	13
<i>Audion Mapping</i>	13
<i>Directed Rendering and Audion Mapping</i>	15
Conclusion	16
References and Further Reading:	17

Introduction

Current gaming systems depend on a CPU and GPU. The CPU is responsible for system control, game control, physics, character movement etc. The GPU is responsible for generating the display.

The GPU generates the display using a display technique called rasterization, this involves working out where an object is in relation to the player and drawing it. Unfortunately this method does not produce shadows and reflections so these have to be computed and drawn in. This technique has been used in the production of graphics for many years but producing highly realistic images is a somewhat inefficient process and requires a great deal of work on the part of developers.

There are alternative image rendering systems called photon mapping and raytracing which allow better quality images to be produced than rasterization, mainly because the lighting is much more accurate. In addition to better quality images, these rendering techniques require much less work from a developers perspective, accurate reflection, refraction and shadows are effectively generated "free" without the need for programming. Unfortunately raytracing and (especially) photon mapping are highly compute intensive and it has not been (and is still not) possible to use these techniques in real time at a high resolution.

Modern processors such as the Cell have already shown that they can be used for generating raytraced graphics [**RTCCell**]. Raytracing however is highly complex and doesn't map perfectly onto the Cell's hardware. In early attempts the local store has been found to be too small, some of the involved algorithms are not very well suited and other algorithms require significant management overhead.

An 8th generation console will not only need better graphics but better physics, Ai and more. If the CPU is tied up generating the graphics these other elements may suffer. We can however assume CPUs will become more powerful. If the 32nm / 22nm nodes are targeted a Cell with up to 64 SPEs should be possible, there will be no lack of raw computing power.

If a system is designed to photon map and raytrace imagery, a traditional GPU will likely not be present, the graphics will be completely generated by the CPU. However, current raytracing on Cell is relatively low resolution and low complexity, games are high complexity and even today some are produced at the highest HDTV resolution. We may be looking at UHTV in 5 years time, even the sort of computing power 64 SPEs can provide will not be enough.

According to existing figures [**RapidMind**] while raytracing on Cell is possible the complexity, number of rays and resolution is rather limited. Producing a 1080p resolution image with 10 rays per pixel will take a 3.2GHz Cell in the order of 5 seconds. To produce the same quality of image at 60 Frames per second requires 300 times greater computing power - just to render the image.

Performance will no doubt get better and indeed some figures are already considerably better [**RTCCell**]. However this is unlikely to be sufficient for a games system today never mind what will be required in 5 years time.

Photon mapping is even harder, it uses a form of "backwards" raytracing which adds another high compute stage to the rendering which is rather Cell unfriendly. It looks very likely that a hardware accelerated solution will be required to use these techniques to generate images for a games system.

Aside from increasing hardware performance, another solution is to modify the algorithms used in order to make them more cell friendly. This paper describes a system called directed rendering which changes both raytracing and (especially) photon mapping into more cell friendly problems. the use of this technique along with faster hardware should mean photon mapping and raytracing will finally become a viable solution for generating high quality images in real time.

Making Photon Mapping a Cell Friendly Problem

Photon mapping is really two techniques, photon mapping itself involves bouncing rays around a scene to determine luminosity at different points. The second technique involved used is raytracing, this bounces rays around a scene to determine what it looks like. The data generated by photon mapping (the photon map) is used to determine the lighting in the raytraced image.

Bouncing rays around a scene involves jumping pretty much randomly through a data structure describing the scene. This is a highly unfriendly to microprocessors in general and Cell in particular.

If the data structure is small enough the bounces will all be in local store (or stores) on chip, this is much better than going to RAM but it is unlikely the complete data structure will fit on chip. Furthermore additional data has to read and computed, both texture filtering and shading require reading and computing data for each bounce. At 1080p60 the raytracing phase alone may involve working through more than a Gigabyte of texture data to produce a single frame. The amount of data consumed is not the main problem however, the problem is the reads are non coherent, and non predictable.

The solution I propose to this is called directed rendering. The idea is in essence pre-computing parts of what needs to be processed and directing like sections to the same SPE/s for processing. By removing serial random memory access, processor stalls are minimised. By ordering operations coherence is maximised and thus memory bandwidth usage will be highly efficient.

It is best explained with a simple example:

A pair of SPEs are generating a part of an image with two objects in it. When it comes to texturing phase, in order to generate the required image, both textures will need to be loaded from different parts of memory. In a directed renderer objects are sorted so each SPE gets only one object, the rays are then "directed" to the relevant SPE for processing. This enhances performance since each SPE only needs to read textures for a single object reducing memory bandwidth and contention.

Directed rendering will be a complex dynamic process, with numerous different parameters being changed to suite the scene. Constant evaluation of the scene and the renderers performance will be required to allow rendering parameters to be dynamically changed ensurign best performance.

Directed Rendering

For raytracing and photon mapping we break up the problem by computing the bounces individually, a low resolution version is first computed then the results are sorted and directed to SPEs for high resolution processing.

Local Map Generation

The first phase is to work out where the user is, that is, the locality of the user. The vast majority of what a user will see is likely to be in this area.

The area necessary to be represented can be determined by producing a 360°, low resolution raytraced image. This will indicate what is needed in the local scene, some bounces are needed to ensure objects not directly visible are also present. The view is 360° because even though something behind cannot be seen by the player it may have an impact on the lighting of the scene.

The generation of the pre-render will indicate all the local data necessary to create a “local map”, this is a data structure which defines the locality of the user. The local map is divided up and split among a number of SPEs.

A “Low Resolution Local Map” (LRLM) is also created, this is a low resolution version of the local map intended to fit into a single LS. This can be generated by replacing complex shapes with boxes and other simple primitive shapes. The LRLM should be sufficiently small to fit inside a single SPE.

Saving Memory With Geometry

The geometry of a world should be allowed to use double precision floating point so very large gaming areas are possible. The full range will not be visible at one time so while it is necessary for a large world it is not necessary for rendering. The rendering can all be performed using 32 bit resolution for geometry, this is beneficial not only for performance but also for memory usage. If the LRLM uses 4 bytes instead of 8 bytes for resolution the size of the LRLM is halved and more likely to fit into a single SPE's LS.

Obviously for 64 bit data geometry to fit into 32 bits it will need to be scaled and converted. The amount of data is relatively small so these operations will not need a great deal of computing power, they are also parallelisable so if necessary can be divided across multiple SPEs. The same scaling should be possible for high resolution rendering, distance rendering and audio.

Once the LRLM is created the photon map generation computations can begin.

Photon Mapping

The first stage is to compute the first rays. In photon mapping these are light rays which originate from light sources. The LRLM is used to work out roughly which object a ray will hit and where.

Once this is computed the ray is directed to the SPE which holds the relevant part of the full resolution local map, in this SPE the exact landing point is calculated.

We will now have a list of rays and we will know exactly where they hit which object. We can then calculate and load what parts of what textures are used, these can be loaded along with the relevant shaders. Photon Mapping will not need full blown texture shaders as it does not generate an image, however information from the surfaces it hits is used as it is used to calculate things like colour bleeding and the direction that secondary rays bounce. Secondary rays are generated at this stage and rays can also be stopped.

The result of these stages is a list of photons to be stored into the photon map and another list of (secondary) rays. The photon map sorted and stored, the ray data is sent back to the first stage.

The same process is then repeated for the same number of times as the maximum ray bounces permitted or while there are rays to bounce.

The final result will be a complete photon map which is used to determine the brightness of pixels in the ray tracing stage.

Raytracing

If the local map or LRLM is too big, non-visible objects can be trimmed at this stage as they are no longer necessary. This may require some re-organisation but the advantage is the visible full resolution local map can be split across more SPEs than before. If the LRLM is too big for a single SPE it will have to be split, removing non visible data may allow it to be placed into a single SPE increasing performance.

The first stage of raytracing is to compute the first ray for every pixel. The LRLM is used to work out roughly which object it will hit and where.

Once this is computed the ray is directed to the SPE which holds the relevant part of the local map, in this SPE the exact landing point is calculated.

We will now have a list of rays and we will know exactly where they hit which object. We can then calculate and load what parts of the photon map are used and which textures, these can be loaded along with the relevant shaders. Secondary rays are generated at this stage and rays can also be stopped.

The result of these stages is a list of image data for inclusion into each pixel and a list of secondary rays. The image data can be sorted and stored, the ray data is sent back to the first stage.

The same process is then repeated for the same number of times as the maximum ray bounces permitted or while there are rays to bounce.

After all the rays are calculated the result will be a series of sorted lists of image data "dots". These are loaded and computed into the final pixel values.

Rays should be kept in some sort of order at all stages to prevent the need for a mass sorting at the end.

Updating The local Map

Once the raytracing is complete the next stage is to execute the game control, audio, physics and other modules. This may involve the player and other objects moving. The next screen to be generated will thus most likely be slightly different than the previous one.

Any changes to the game will be made to the high res world map but the local map and the LRLM will both need to be updated to reflect these changes.

This is not only necessary for directed rendering but the update process can be used to remove stored Local Stores (see below) which contain texture or other data when that part of the object is not going to be displayed .

Issues

Rays Hitting Irregular Shapes

An issue with boxing irregular objects means the hit point of any rays hitting the box may be incorrectly calculated as hitting the object when in fact it misses it.

A solution to this is to use additional boxes to enclose the areas where this will happen and perform an additional check when these boxes are hit. If the ray is found to hit the irregular shape it is treated as normal, if not a new ray is generated with the same direction as the first ray. This ray can then be calculated as a secondary ray against the low-res local boxes.

Local Map Data Too Large To Fit Into A Single SPE

If a section of the local map data is too large to fit into a single SPE the solution is to dynamically adjust how the accurate data is distributed. How it is distributed is based on the number of rays hitting an area. If the number of hits is low the data should be split up and streamed in.

If the number of hits is high an alternative method should be used, the data is subdivided over a number of SPEs for computation.

In both cases, to facilitate this, the ray hits can be sorted so they are in the same sequence as the data being streamed in.

There is also a third option if subdivision or streaming is not possible, in this case a software cache can be used.

Inaccurate Ray Hit Point Estimation

If rays hit a box from a steep angle, hit points may be estimated incorrectly if the surface of the object is not right beside the edge of the box.

This will not be a problem if the SPE holds all the data, but if this is not the case and data is streamed it could be problematic.

Depending on the complexity of the shape, it may be possible to estimate the error and attempt to correct for it. Alternatively, if this cannot be done the accurate hit computation could use a cache instead of trying to stream the data through.

Rendering Distant Objects

Distant Photon Map Generation

Distant objects will not fit into the local map and will need to be handled in a different manner. The parts of the scene outside of the local map are instead rendered from a smaller version of the full “world” map called the low resolution full map (LRFM). To avoid excessive computational overhead complex objects are represented by simplified versions built with normal triangles (or perhaps NURBS). Full detail is only necessary at close range so there will be no need to do the secondary high resolution object intersection stage. There will however still be 2 stages, a pre-render will be needed to indicate what part of the full map is visible. This can be broken up into sections and the data and rays directed to the relevant SPEs.

The relevant parts of the LRFM should be loaded completely into SPEs if possible. There is a distinct possibility this will not be possible in which case streaming the data will be the preferred alternative. In order for streaming to work the map will need to be divided into chunks which represent a specific area. The chunks cannot be too large however as at least 2 will need to fit into an SPE at one time (for double buffering).

The amount of data required by the LRFM can be further reduced. Objects can be removed completely by changing them into images on textures. Instead of a window with sides, ledge and glass, the window can become an image on a texture. The lack of geometry will be unnoticeable as the building is far away. The reflection from the window will still be possible however if a texture indicates that part of the building will reflect light.

For even larger scenes even a low resolution map may be too big to use. In this case textures can replace the geometry giving the impression of being able to see far in the distance.

Photon Mapping Large Areas.

For distant lights the number of rays and ray bounces can be reduced to reduce the computational requirement. This could have the side effect of making objects in the distance darker than they really are so the photons used should have a larger value to make up for this. Lower numbers of photons also means lighting can be erratic and this may show. The “radius” read from the photon map can also be increased to make up for this, a default lighting value may also be useful if the number of photon samples becomes unreliable.

Distant objects will be subject to aliasing problems when the scene is raytraced, all distant scenes will need to be antialiased by shooting additional rays. To offset the overhead of the additional rays, they have a reduced level of bounces or none at all. A large building will reflect a great deal in its windows, a single bounce will suffice for this.

Measuring Distance

Initial rays in the pre-render can be used in order to determine the distance of objects. A virtual lens can be added for effect, however if this lens is modelled as an object it will defeat any attempt to measure distances. This will need to be taken into account and secondary rays used to indicate distance.

Antialiasing

Aliasing is a problem with all graphics systems and often requires large amounts of computations to remove or reduce. In the case of raytracing, antialiasing can be achieved by oversampling. It is not necessary at all points on an image however, aliasing is most noticeable around object edges.

The directed renderer should be able to assist antialiasing when it is creating the low resolution local map. If the boxes made so they are not an exact "fit" to the accurate objects, there will be misses around the edges of objects triggering secondary ray generation. This is useful as it is the edges of objects we wish to find, additional rays on both sides of the object boundary can be generated to facilitate antialiasing, these can be collected in the final pixel rendering phase to generate the antialiased pixels.

Additional rays generate additional computation overhead so if a large number are generated this could slow the renderer. The solution to this is to reduce the number of jumps or sub rays these extra rays can make, this will reduce the computational load. Assigning a proportion of computing time to antialiasing will ensure the computation does not take longer than expected.

Non-edge Antialiasing

There are 2 other areas in images antialiasing may need to be applied: objects at a large distance and textures.

Objects at distances are subjects to aliasing because adjacent pixels can be displaying parts of different objects. This can be seen in many images where squared floors go into the distance, it can also be seen in the sunflowers video **[Sunflowers]**, the sunflowers in the distance turn into a mass of random dots.

There are a couple of possible solutions to antialiasing distant objects:

The first method would be to do as is done with edges and shoot additional rays, in this case the number of ray bounces and secondary rays can be reduced sharply as they will provide little useful information at a distance. Ultimately the number of ray bounces could be dropped to 0, raycasting the distant view.

Another method that could be used to antialias distant objects would be to apply a low pass filter (aka blur) the pixels generated together. This is an undesirable method to use however there may be instances in which even oversampling will not be sufficient. Blurring could be very noticeable so its use should be subtle.

Antialiasing Textures

Textures are not always antialiased and this is sometimes noticeable. Obviously shooting additional rays will create too large an overhead since this effectively means oversampling every pixel.

One method to antialias textures is to oversample at the texture filtering stage. This effectively means using more texture samples than necessary to generate the final texture sample used.

Another method would be to blend adjacent texture samples as they are generated. This requires the relevant rays to be processed together but this should not be a problem as this is desirable from a coherence point of view anyway. Blending can of course be applied at the final image generation stage.

Temporal antialiasing

A "cheap" method of getting improved antialiasing is to antialias across time (this has been used in some ATI cards). On one frame a specific pattern of antialiasing is used, on the next frame a different pattern is used. The different antialiasing on each frame produces subtly different images but because the frames are displayed so quickly the eye blends the two images into one effectively doubling the amount of antialiasing.

Temporal antialiasing has the disadvantage that frame rates need to be locked to the display rate of the display. If this is not done it is possible that not all frames are fully displayed and thus both the different patterns are not displayed defeating the effect.

Use of Objects Instead of Triangles

Unlike rasterization, raytracing does not require objects to be constructed from triangles. This has advantages in memory usage and reduces the complexity of object trees.

For example, a detailed sphere may be made up of thousands of triangles, each of these has 3 vertices requiring 3 points (x,y,z) to describe each one. If the geometry uses 64 bits per point that's 72 bytes to describe a single triangle. For a sphere made up of 1,000 triangles that's at least 72,000 bytes to describe it (less with compression).

A sphere "object" on the other hand requires a single x,y,z position, and a radius to describe it - just 28 bytes.

Modifying objects will take considerably less computation time as less is required to be modified. If a sphere is expanding or contracting every single triangle has to be modified, same if the sphere is moving. In the case of the object sphere only the radius has to be changed to change its size and only 3 co-ordinates are modified if the sphere moves.

The use of objects means object's shape will need to be calculated exactly in order to work out the exact position of a ray hit, this imposes a computational overhead. This overhead has to be weighed up against the additional bandwidth and tree searching time required for an object made up of triangles.

In reality objects will be rather more complex than spheres, however the use of objects instead of just triangles will allow more organic and realistic shapes to be created. Additionally in the case where a mathematical description becomes too complex, there is nothing to prevent an object being made up of triangles.

Procedural Objects

Procedural object generation is already used in games to ensure in-game objects are not all the same, indeed it is considered so important that the Xbox360's CPU was designed in part to accelerate this specific technique.

In the case of raytracing the procedural objects may never need to be generated as triangles. Instead it may be possible to use the procedures directly to compute ray intersections. In this case the objects will also be objects in programming terms, they will contain procedures to be called. Intersection for primitives should be handled by the renderer but if more complex shapes are used an object may contain the code to perform the intersection.

It will be highly complex to manually write an object in this manner but this should be possible with a tool. The tool should be able to decide when the mathematical overhead will be too great and generate a triangle based object instead. If the object is performing the intersection it should also be capable of generating the low resolution boxed version used in first stage rendering.

Dynamic Object Deformation

Triangle based objects can only be deformed along the lines of the triangles, if the object does not contain a large number of triangles breaking the object into pieces is likely to be unrealistic.

With non-triangle based objects breaking apart can be controlled by an algorithm which creates sub-objects. Objects can contain physical parameters which can be used by the physics engine to control the break-up of the object, thus the break-up can both look and act correctly.

Prior to breaking up the object can remain simple and thus use little memory. This is more difficult with triangle based objects. For example consider a brick wall which is hit by a tank. In order to break apart correctly, the bricks will have to break apart separately. For a triangle based system this means the wall will have to be made up of dozens of triangles irrespective of whether it is hit or not. In an object system the wall can be a single object until it is hit, additional information can store the location of the bricks (this can be generated from a formula). The wall is simpler and more flexible as an object rather than a collection of triangles.

The use of objects instead of triangles offers opportunities for saving large amounts of memory whilst simultaneously allowing highly complex shapes. Objects will incur a higher computational cost than triangles and this requires code however both will require less bandwidth than large numbers of triangles and the searching thereof.

Texture Processing

Textures are used to indicate the properties, these properties determine what the final object looks like. The object could be made out of metal, stone, water, glass, mud, plastic, wood or any other number of materials. In the case of raytracing they can also be used to determine when secondary rays are generated and where they go. The photon map will also be read at this stage to determine the brightness of the generated "dot". Multiple dots will be generated by the different secondary rays and these are combined into the final pixel for display.

Processing textures is generally considered a Cell unfriendly problem, however the same directed processing technique can be used to make texturing into a predictable problem which is very Cell friendly.

After each stage of raytracing the system will have produced an accurate list of points in space, these all need to be processed before the next stage can begin. Because the rays are sorted before accurate hit positions are determined we can say all rays in a given section will hit in a pre-determined area, we can thus calculate which parts of which textures are required.

In order to make the processing more efficient the ray hits can be sorted into sub-sections so they are also in order, the textures are then be loaded in the same order. Data loads can also be double buffered in the LS so once one section is processed the next section can begin processing immediately. This is how texturing can be made into a problem well suited to the Cell.

Texturing Multiple Objects

The accurate ray hit processing will likely include multiple objects. The sorting of the rays needs to ensure all rays in each object are together, this will ensure coherence in the texture processing.

It is also quite possible that an object will take up a large amount of on screen space and thus the same object could have rays processed by a number of SPEs. In this case the texture processing for that object should be sorted into different areas of the object and directed to specific SPEs. This will not only ensure coherence but also ensure that the same data is not loaded into different SPEs (except for overlaps).

Ideally a single object should not be assigned across multiple SPEs unless it is taking up a large amount of ray hits. If the object is relatively small but has been assigned across multiple SPEs the processing of that object should be directed to a single SPE.

Optimising Texture Data Layout

To improve performance of loading textures from memory they should be organised in memory so loading a square block can be done in a single read.

To process an object many textures may be required, these could all be organised as squares and placed adjacently in memory. In this case reading all the data from the different textures required can be fetched in a single read.

When an object is displayed on screen there is a good chance most or all of it will be displayed. The texture squares to be loaded should thus be large rather than small. This not only reduces the number of loads required but larger blocks of data get better use of memory bandwidth.

Optimisation Of Texture Loads Across Frames

Even when there is fast movement it normally takes several frames for an object to move out of view. This can be taken advantage of by saving the contents of a LS after it has finished texture processing. When the next phase of texture processing is began the SPE allocated with that area loads the data saved and proceeds to process the ray hits in the reverse order. The reverse order has to be used because the data stored will have been used by processing ray hits at the end of the ray array. By saving and loading data like this the maximum bandwidth of memory will be used, this is more efficient than re-loading the different texture blocks separately.

Texture Data Lists

Another method of saving bandwidth is to save the list of textures loaded. Since most of the next frame is likely to be very similar most of the data in the list will be needed again. When the list/s are reloaded they can be used to speculatively load texture data before it is needed. This can be initiated before any texture addresses are calculated thus giving a processing boost.

Texture Compression & Mipmapping

Texture compression and mipmapping is already commonly used to reduce bandwidth usage. This should of course also be used alongside the other techniques I have described. If an object is further away mipmapping should also be used to minimise bandwidth usage. There are many techniques used in rasterization which will be useful in directed rendering.

Fractal textures

One method of making objects more realistic looking is to use fractal mathematics in the dynamic creation of textures. Fractals have the property that they become more detailed no matter how close you look at them. Generating textures on the fly means you do not need to have a large high resolution texture sitting in memory all the time. This will require additional computation time but as ever the cost of this needs to be weighed against the memory and bandwidth a high resolution texture will require.

A compromise may be to use fractals to generate the high resolution texture only when it's needed, the memory can be freed when it is no longer necessary. Another advantage of fractals is they are different depending on initial conditions, a single algorithm which generates textures for bricks can be used to generate vast numbers of bricks all of which will be slightly different - just like real life.

Spherical Texture Filtering

Texture filtering is used to prevent aliasing in textures, it is also used to prevent textures distorting or showing artefacts. The filtering systems in GPUs are designed to sample and filter textures on triangles. An object based system is likely to have textures on curved surfaces at arbitrary angles to the viewer, a GPU type filter is not designed to filter textures for this sort of shape so another type of filtering is required.

For a curved surface the texture sampling and weighting needs to be designed to fit onto the curved shape. These should ideally be calculated on the fly and the texture points filtered into a dot accordingly. This is highly compute intensive operation and given multiple dots are combined to create a final pixel, it is repeated more than once per pixel. This is a highly bandwidth intensive operation so any optimisations which reduce computational or bandwidth overhead should be utilised.

Pre-computing the sampling and weighting is likely to be necessary, these pre-sets could be held in the LS and used as required. Directed texture processing should ensure texture is in the local store before use so memory bandwidth and compute power should already be optimised.

Raytracing Audio

Raytracing techniques can be used to simulate the path of audio in a scene creating a realistic sound field. Audio however is different from lighting in a number of ways, in an audio system all of the following have to be considered:

Sound travels with a finite speed, if an action is far away the sound will arrive after the action is seen. Even though a sound can be very short, it can continue to bounce around a scene for several seconds. The movement of a player changes the pitch of the audio, moving towards a sound makes it's pitch rise, moving away makes it's pitch fall.

The brain is highly sensitive to audio, at 44KHz even a single wrong sample is noticeable.

The brain is also sensitive to delays between actions and sound, if the sound is heard more than 10-25ms after the event is seen a delay will be perceived. This is realistic if the action is far away, not if it is close.

To generate an accurate sound-field a technique called audio raytracing can be used. However, if this technique is similar to image raytracing it may not take full account of movement or the speed of sound.

Audion Mapping

I have devised a new technique in which audio raytracing is combined with a photon mapping like technique called "audion mapping".

4 steps are used:

- Audion mapping (pronounced as "audi on")
- Audio raytracing
- Sound construction
- Sound re-dispersal

Audion mapping

This is much the same procedure as photon mapping but is done for audio. The LRLM can be used for this purpose as exact reflections off complex shapes can be ignored, it's important to note however that the version of the LRLM used in photon mapping should be used for this rather than the later "visible" version the raytracing phase can use.

Each sound source generates sounds rays which are bounced around the scene depositing "audions" onto an "audion map". An audion will need to indicate what the original sound source was, the bounce number, filtering characteristics, attenuation characteristics and the total distance travelled. The filtering and attenuation is additive so this will be added each bounce. Alternatively the audion could just indicate the material it's hit and allow the filtering characteristics to be computed at the sound construction stage. An additional flag can be used to indicate direction, that way sounds travelling away from the player can be ignored.

The number of audion rays initially needed to be sent out is relatively low but at each bounce there will need to be a number of secondary rays created. Since sound tends to go in all directions these will be broadcast randomly. The total number of bounces will be fairly low since sounds moves much slower than light and attenuation will eventually make sound inaudible. Sounds in a small space with relatively little attenuation will however create large numbers of sound rays.

Because of the relatively slow speed of audio, it is possible that a sound ray will not hit the player or any other object. In this case the audion will need to be deposited in mid air and marked as such.

Audion rays are sent out for each audio source, sources can be indicated by the game engine (e.g. gun firing, engine running, bullet flying) or by the physics engine (e.g. bullet hitting object, object hitting object). Constant sound sources should spread their rays slightly randomly so unnatural resonances are not set up.

Audio Raytracing

As with image generation, once the audio is dispersed around the scene, it is raytraced to indicate what can be heard. In this case the purpose of the raytracing is to locate the audions, this can be done in much the same way as with the photon map with all audions within a certain radius of a ray hit detected.

Unlike in image generation, rays will not need to bounce much (if at all) and will need to go in all directions instead of just the line of sight.

The distances to audions and the distances between audion bounces are important as these indicate delays. The total distance travelled and the time of sound emission needs to be calculated, if the distance is too great the sound cannot be heard and the audion is either ignored or a delay added before it can be heard. The position is also important because the ears filter sound differently depending on where it is coming from.

Sound Construction

The final sound produced is a composite of all the sounds described by the audions and any direct sound sources. The sources are filtered, attenuated and delayed.

When a player is moving sounds shall change depending on the direction and speed of the movement. This also needs to be considered before the final mix stage. Since audion mapping will be done at discrete intervals any changes in pitch or attenuation will need to be made gradually over the length of the final sound produced. Any abrupt changes will be highly noticeable and sound bad. By making changes gradually sounds will become louder gradually as you move towards them as they would naturally. Also as something flies past you the pitch as well as the sound level will have to change.

At the final stage of sound construction, the sounds are mixed into a group of sound channels. The number of channels depends on the desired output type, stereo, 5.1, 7.1 etc. For 2 channels virtual surround type technologies can be used to enhance the directionality of the sound.

Sound Re-dispersal

In order for the sound of a scene to be realistic it will need to keep bouncing sounds around the scene even after the the listener has heard it them. All audions will need to be evaluated to see if they will continue, if so they will restart their journey where they left off.

As audions travel they will attenuate with each bounce, eventually the attenuation will mean the sound becomes inaudible, the attenuation level can be tested and if too great the audion can be deleted. The sound level can also be compared with what the player is hearing and if below that threshold (at all frequencies) the audion can be dropped. An exception to this is when the sound level suddenly drops and the deleted audion would have been audible, however this can be tested for.

External sounds

As with the raytracing and photon mapping above, large areas are treated differently from the relatively small "local" area. With audio, traditional techniques such as echos and reverberation effects can be used to simulate large areas in a convincing but non exact manner. The low res full map can be used to generate information about the audio characteristics of large areas, this information can then be combined with traditional techniques to produce more accurate representations of the sound. The traditional techniques in this case are replacing the audion mapping.

External sound sources should be modelled using these techniques to make them more realistic. Distant gunshots or bombs have distinctive sounds, including them will add authenticity to a game. If a bomb goes off not far from the building you are in the building will rattle and this makes a discernible sound (something I have first hand experience of!). Adding details like this will add authenticity to a game.

Resonances

This is not described in the technique described above but should be considered. We all know the sound of something when it is dropped. A bottle dropped on carpet makes a different sound from a bottle dropped onto wood or stone. A fully accurate simulation of these sorts of sounds could be done but this is too compute intensive and unnecessary for games. There are "light" simulation techniques available which could be used to good effect.

When combined with audion mapping simulating resonances could create a highly convincing soundscape.

Directed Rendering and Audion Mapping

Directed rendering will also be applicable to audion mapping. The 2 stage rendering and ray sorting should not be necessary as the LRLM should fit into a single Local Store. However this may not be so in which case the LRLM will need to be divided across 2 or more SPEs. Rays will then need to be sorted and directed to the relevant SPEs.

If the scene makes use of objects rather than triangles, it will be useful to split the objects up then direct rays to the SPE handling the relevant object.

Audio mapping will be highly parallelisable so the workload can be split across multiple by coping some or all of the LRLM. The Sound construction stage will also be parallelisable so filtering and delaying duties can also be split.

Conclusion

In this document I have described a technique of directed rendering which should accelerate the performance of raytracing and photon mapping significantly on Cell BE.

Other techniques are also discussed (e.g. reducing detail in the distance). Many of these or similar techniques are already in use in rasterization based systems but will also be useful in a directed rendering system. In some cases reducing the level of detail will be necessary to keep compute requirements in check (e.g. distance rendering).

The technique may enable the possibility of real time raytracing in next generation games consoles, before that it will be applicable to other rendering systems. It may even be possible using current gaming systems but the GPU will likely be required to assist in parts of the work (e.g. texture filtering and processing).

The technique of breaking tree traversal into 2 parts and executing traversal of rays grouped by level may also be applicable to other domains than raytracing. This however will require further investigation.

Directed rendering may also accelerate rendering on conventional processors but this has not been looked at. It should be possible but it will require careful control of caches.

I have not developed any code for this problem, this is expected to be a complex task and many more issues will need to be addressed which have not been discussed here.

About the Author

Nicholas Blachford has written widely read articles about Cell and other technology subjects. He is currently learning more about Cell development.

This paper was written primarily as a part of a proposal for a next generation gaming system (PS4). It was also written because he considers directed rendering to be an interesting potential solution to an interesting and complex problem which may be applicable elsewhere.

Note: He has no training in and has not worked on graphics so please forgive any glaring errors!

References and Further Reading:

[Photon Mapping]

<http://www.cs.mtu.edu/~shene/PUBLICATIONS/2005/photon.pdf>

[RTCell]

Ray Tracing on the CELL Processor

<http://graphics.cs.uni-sb.de/~benthin/cellrt06.pdf>

http://www.sci.utah.edu/~wald/RT06/papers/cell_vortrag_final.pdf

Real time raytracing of Quake 4

<http://www.q4rt.de/>

[RapidMind] RapidMind Cell raytracing

http://www-03.ibm.com/industries/media/doc/content/bin/RapidMind_Raytracer_Demo.pdf

[Sunflowers]

<http://www.intrace.com/gallery/sunflower.php>

Audio Raytracing

<http://jerry.c-lab.de/~wolfgang/icmcs99.pdf>

[SaarCOR]

Raytracing Hardware

<http://graphics.cs.uni-sb.de/SaarCOR/>

<http://graphics.cs.uni-sb.de/Publications/2002/Schmittler-AHardwareArchitectureForRayTracing.pdf>

<http://graphics.cs.uni-sb.de/~woop/rpu/rpu.html>