Real-Time Ray Tracing Methodology

J.D. Bruce
Rev 1: April 2016
www.j-d-b.net

INTRODUCTION

The overall goal with real-time rendering is to create an engine which has a consistently high frame rate even with a large number of objects in the scene. One example we will briefly examine is Euclideon's "Unlimited Detail" engine, which can render large point-cloud scenes very quickly. On their website they claim that polygon rasterization engines are becoming obsolete and that point-cloud engines are the future. They claim polygons are starting to become smaller than a single pixel in modern games and so there's no point to not using point-cloud data[1].

At first glance this logic may seem pretty solid, but it breaks down when you seriously start to think about it. There are several crucial disadvantages when it comes to using point-cloud data for a game engine. The first problem is the huge size of point-cloud objects. Although many polygons in modern games are smaller than a single pixel, many large surfaces are flat and can easily be represented by a polygon with a small number of vertices. However, to make a large surface in a point-cloud format you need a large number of points and it's very inefficient.

Another problem is that animations become harder and more demanding when the object contains a large number of independent points instead of vertices which are connected together and form surfaces. Shifting a single point in a point-cloud object will leave behind a hole where the point previously existed, but with a polygon object the surfaces simply stretch and warp when vertices are moved around. For all these reasons and more, polygons are the best way to represent 3D objects in a game engine, and it will remain that way for the foreseeable future.

Euclideon says their engine is so powerful because it doesn't matter how many points are in the scene, their engine will quickly locate a point for each pixel, allowing the scene to be rendered without a GPU. They claim the trick behind this is to put the point-cloud data into a database, allowing a very fast search to be performed which finds the closest visible point for each pixel. This may almost seem like magic, and might make point-cloud rendering seem very attractive, but there are very important reasons this will not work for video games.

Again, it becomes very difficult to do things such as animation when you need to manipulate a database in order to update the points in the scene. So far Euclideon has only demonstrated static scenes using point-cloud data which is scanned in from the real world. For example they have a demo where you can explore a static church scene and another demo where you can explore a static forest scene, they are both approaching the level of photo-realism yet render in real-time. Sadly, games are not static scenes, objects move around and things change.

Another problem with using a database of points scanned in from the real world, is that you lose distinction between objects. Everything in the scene is simply represented by a large bunch of points stored in a database. So if a specific object needs to be manipulated, it's impossible when you cannot tell which points are associated to which objects. It may be possible to use this technology in a game but each object would need to be scanned separately and the database would need to be designed to handle separate groups of points for each object.

Based on the fact Euclideon has only released demos of static scenes it's probably fair to say they haven't cracked any of these important problems, even though they have long claimed their technology could be applied to games. Taking into consideration all these facts, a reasonable conclusion is that we should stick with using polygon meshes for games. At least that way we will be able to use existing polygon meshes and we wont have to make everything from scratch. Now let us explore some non-traditional methods for rendering polygons.

ACCELERATION STRUCTURES

Traditionally, polygons scenes are rendered using a process commonly called rasterization. It is a very fast method but it has several limitations compared to ray tracing, for example shadows and reflections are buggy and unrealistic. However, ray tracing has a reputation for being very slow and there's a reason it's not used for real-time applications. Ray tracing involves shooting out a "light ray" through every pixel on the screen into the scene, then finding the closest surface to intersect with each ray, then finding the color of the surface for each intersection.

Using that method we can render a 3D scene but for every ray we shoot out, we need to loop through every object in the scene to check for an intersection. If we have millions of pixels and millions of objects in the scene, which is possible for large games, then we're talking about a huge number of operations. The logic behind using ray tracing is based on the hope that we might be able to achieve real-time speeds if we can dramatically reduce the number of operations necessary for each ray, allowing the color for each pixel to be quickly discovered.

In order to make it work at real-time speeds we obviously needed some sort of optimization technique. A common tactic is to use some type of acceleration structure such as an octree or kd-tree to reduce the number of ray-polygon intersection tests[2]. The tree structure could also be thought of as a type of database structure, allowing us to more quickly locate objects in a region of space without having to search through all the objects in the scene. Of course it's not the same as having a separate database server to handle search queries but the concepts are related.

The way most tree structures work is to divide 3D space into cubes or rectangles and use that to sort the objects in the scene. By doing that, we don't need to check all the objects in the scene when we shoot out a ray because we can ignore objects inside any cube the ray doesn't intersect with. There are however a couple of issues with this approach. If objects are moving around, their position in the tree structure will change and it may become expensive to update the tree each frame if there's a large number of dynamic objects. A much smarter approach is to precompute a separate tree structure for each static mesh, which is perfect for real-time ray tracing.

Another large problem with using tree structures on whole scenes is that large open scenes will be slow to render because it will take longer to find intersections. If the ray doesn't hit any obstacles and ends up hitting the sky, that would generally require many ray-cube intersection tests because the ray traveled such a long distance. In many game scenes, a great deal of the sky is visible a lot of the time and other objects can be very far away. If each cube is $1m^3$ that's 1000 cubes over 1km, so an octree probably isn't a very great solution if we want real-time rendering speeds.

The octree structure is nice because we only have to check objects inside the cubes along the path of the ray and we can stop when we find an intersection, but some times it takes a very long time to find an intersection. What we want is a method where the distance of objects doesn't affect the search time. Remember that the real goal is to find the closest intersection for each ray as fast as possible. We need to think of it like a search algorithm and we want that algorithm to have a fast and constant search speed like the Unlimited Detail algorithm.

2D ACCELERATION STRUCTURE

The following method is probably not a novel one, but it's easy to implement and arguably one of the best approaches. Rather than use a 3D acceleration structure, we use a 2D acceleration structure. The structure is designed to order the objects according to what pixels on the screen they are likely to cover when rendered, by using the bounding boxes of the objects. For every object, we use the simple 4-sided bounding box to figure out which pixels might be covered by that object. The object is added to the list if any pixels are covered by the bounding box.

Well it actually shouldn't be a list of objects for every pixel, because that would basically be like rasterizing the bounding boxes of the objects to the screen, which wouldn't be very efficient because objects often cover many pixels and we would end up with the same object in many different lists. Instead we break the screen up into larger "blocks" which contain a number of pixels. If the bounding box of an object covers any of those blocks, we add it to the list for that block, at a position ensuring the objects are ordered by distance.

Then when we shoot a ray from a pixel, we only have to check a small number of objects against that ray because for every pixel we have a list of objects likely to intersect with a ray sent from that pixel. First we check that the ray hits the bounding sphere of the object before we actually check the polygons in the object. Essentially, the bounding box gives a rough estimation of what pixels the object will cover, then the bounding sphere further enhances our knowledge of how likely the ray is to hit the object. We can skip all the polygon tests if any previous test fails.

If the ray hits the bounding sphere it is very likely to hit some part of the object but you may be wondering why we don't also use the bounding sphere to determine which pixels the object might cover. The reason is because that's actually much trickier to determine since a sphere will be distorted into an ellipse when it's not directly in front of the camera. Although the bounding box doesn't bound the object as tightly as a bounding sphere, it seems to be a much easier and maybe even a faster approach since there's less math involved.

Figure 1a

Black area shows pixels where object list was empty so no ray fired
Blue area shows pixels where ray didn't hit object bounding sphere
White area shows where bounding sphere was hit but mesh missed
Other pixels are where the ray intersected the matchbox mesh

There are several advantages this 2D acceleration structure has over a 3D acceleration structure. It needs to be recomputed each frame but so do other tree structures, however the 2D structure maintains the same dimensions regardless of game world size. Also, the distance a ray travels does not matter with a 2D structure like it does with an octree. Each pixel has a list of objects associated with it, so we simply need to check our ray against the objects in the list for the given pixel. Since the objects are ordered, we can also stop when we find an intersection without having to check all objects in the list.

What this means is that we get occlusion culling almost for free because we can ignore objects behind other objects. We only have to loop through the objects in the scene once to build the 2D acceleration structure, which is a fair price to pay for the speed boost it can give to scenes containing many objects. Instead of having to loop through every object in the scene every time we fire out a ray, now we just check the objects in our list for the pixel which the ray was shot through, vastly reducing the number of triangles checked against each ray.

It seems that the best data structure to use for these lists is the singly linked list structure. In order to make sure the objects in the list are ordered by distance, we need to be able to insert a new entry at any point in the list, and linked lists do that very quickly, where as an array would need to be resized and data shuffled around. We also don't need to access random elements in the list, we just need to be able to iterate through the list from the closest object to the furthest, and a linked list will let us do that fairly quickly.

After we apply the process described so far, we end up with details about the closest intersection point for each ray we shot out into the scene. Then for each of those intersection points, we want to find out the color of the surface at the exact point it hit so that we can do texture mapping. It turns out this is extremely easy when you're doing ray-triangle intersection tests because there are fast algorithms which return the barycentric coordinates of the point on the triangle surface where the ray hit, which can be used to calculate the texture coordinates.

OCCLUSION CULLING

Lets talk about rasterization and how that works for a moment because it's necessary for understanding why this approach is arguably superior. Rasterization of a 3D scene is generally pretty fast because we loop through the objects only once, drawing all the triangles to the screen as we go. The process of drawing the triangles onto the screen is known as rasterization, but usually the triangles aren't directly drawn to the screen. With deferred rendering several buffers are used to store information about the closest surface for each pixel[3][6].

Every object in the scene can be broken into triangles, so for every triangle we map the 3 corners to the screen, then we typically use some sort of line scanning algorithm to determine which pixels are inside of the triangle. For every pixel we find inside the triangle, we check whether there is already another triangle which is closer to that pixel by using a depth buffer. If the depth value for that pixel is empty or is further away, we save information about the triangle we are currently rasterizing to all the buffers, overwriting any previous data.

After doing this for all objects within the field of view, the buffer should contain information about the closest surfaces visible to each pixel on the screen, just like we have after finding the closest intersection for every ray when doing ray tracing. However if we compare the ray tracing method just described with the basic rasterization method just described, then we see that rasterization engines are not naturally good at occlusion culling since the triangles aren't easy to draw in order, so they usually just draw all the triangles in the field of view.

Of course there are some occlusion culling methods used by rasterization engines to avoid rendering objects behind other objects, but they tend to be either fairly ineffective or difficult to set up. The 2D acceleration structure used in the ray tracing method just described ensures that objects in each list are sorted by distance and that means we can stop searching for ray intersections before we check all the objects in the list, which has the effect of culling almost every single occluded object. However overlapping objects will prevent it from being perfect.

The other reason this method could possibly be faster than rasterization is because we don't have to find all the pixels inside every triangle, we just need to know whether or not the given triangle is intersected by the given ray. The main thing which will slow down this approach is the fact we may have to check the same object against many different rays if the object covers a large portion of the screen. One thing we could do to minimize redundant checks is use our 2D acceleration structure to store a list of triangles instead of objects.

Essentially, we would rasterize the scene much like how a rasterization engine would, but without any texture mapping and in a reduced resolution so we don't have a separate list of triangles for each pixel. That should be fairly fast and may be the right way to go, but the idea of using the bounding boxes is still very attractive because it's very easy to find a 2D bounding box which contains the object on the screen when given the 3D bounding box. It also avoids having to loop through every triangle in all the objects, so it may be faster.

TEXTURE MAPPING

So far we've only looked at the first part of the rendering process, where we must find which surfaces need to be drawn to which pixels, and we do it by using ray tracing techniques instead of rasterization techniques. Now we will discuss the second part of the rendering process where we use texture mapping to find the color at the exact point on each surface where the ray hit. As mentioned this is actually very easy, it's pretty amazing just how organically ray tracing and texture mapping of triangles fit together, as if they were almost made for each other.

The Möller–Trumbore ray-triangle intersection algorithm is one of the fastest known algorithms for doing ray-triangle intersection tests and it just happens to return everything we need for texture mapping[4]. It gives us an answer as to whether the given ray intersected the given triangle, and if they do intersect it tells us how far away the point of intersection is from the ray origin and also returns the barycentric coordinates of the intersection point on the triangle, which allows us to easily calculate the exact point of intersection in world space.

The barycentric coordinates (aka UV coordinates) are really all we need for the texture mapping but the depth information is useful for several things, including shading calculations and finding the optimal level of detail required, which we will discuss in the next section. The aim is essentially to recreate many of the same buffers used in deferred rendering rasterization. First we fill all our buffers: depth buffer, normal buffer, diffuse color buffer, etc. Then the information held in those buffers is used to generate the final scene lighting just like deferred rendering.

It would be a bad idea to find the color and normal for every single ray-triangle intersection because even though we have our objects ordered by distance, we may miss some objects, strike several triangles in the same object or several objects may be overlapping and we need to check them all. So we cannot necessarily stop at the very first intersection we find for each ray, we usually need to do a little bit of extra work to make sure we've found the closest intersecting point. Once we are sure our buffers contain correct information about the closest visible surfaces we can then do texture mapping.

If we delay the texture mapping then we must also remember to record the closest intersected triangle and the UV coordinates. The UV coordinates allow us to quickly calculate the XY coordinates of the intersection point on the texture. Each vertex on the triangle is assigned an XY texture coordinate when we create the object. The UV coordinates simply tell us how we should interpolate between those vertex coordinates to get the final XY coordinate which tells us the color at the exact point where the ray hit. It's almost exactly like interpolating the vertex normals.

So to fill the diffuse color buffer we do that for every intersection point saved to our other buffers. Once that is done it finally becomes possible to do pixel shading, but we could also do anti-aliasing by shooting more than one ray through each pixel[5]. For every ray sent through the pixel we need to find the color of the intersection point and do shading on it, then average the final color obtained for each ray to get the final color of the pixel. Not only will that reduce jagged edges but it will increase the fidelity of textures since we have more sample points.

MIPMAP INTERPOLATION

Ray tracing actually has aliasing issues like rasterization and texture sampling can become a problem if the surface is far from the camera or it's being viewed at a sharp or odd angle[5]. This is because multiple pixels on the texture can cover a single pixel on the screen, so taking a single sample is usually not enough. This is where having several textures (aka mipmaps) with different resolutions (aka levels of detail) can become very useful. If the surface is far away then by using a lower resolution texture our sampling will become more accurate.

We want to ensure that the texture resolution is as close to the screen resolution as possible. However that wont work for surfaces being viewed on a sharp angle because as the surface moves further into the distance we lose the resolution correspondence and we still get multiple texture pixels covering a single screen pixel. One nice way to reduce this problem is to interpolate between the different levels of detail for the texture. Using the distance to the point of intersection, we use the two textures which have the best resolution correspondence.

When a ray intersects with a surface, it's unlikely that any of the mipmaps will exactly match the resolution required for the distance to that point. Actually it will be some where between the two best textures, so what we really want to do is interpolate between those two textures. Once these textures are located, we take a sample from both of them and then combine the sampled colors in such a way that they are weighted according to their resolution correspondence. For example we may use a weight of 0.3 for first sample and 0.7 for the second, always summing to 1.

One interesting thing to note about this process is that the resolution of the textures has no affect on the rendering time. When given the UV coordinates of an intersection point on a triangle, converting that into a XY position on the texture image is simple and fast. Given the XY position it's easy to calculate the exact index we need to access in our texture color array. Unlike a linked list, random lookups of an array are fast and happen in constant time, so it's pretty much irrelevant how large the array is, the only limiting factor is memory space.

In other words, given the UV coordinates of a point on a triangle, it's possible to find the exact color corresponding to that point in constant time. This doesn't seem to be true for rasterization engines, and may be related to the way some engines handle sampling or how they transform textures to account for perspective changes. Some sources report that well coded rasterization engines hardly experience any impact at all from using higher resolution textures and there's a few old threads debating this issue but it really comes down to how the engine works.

Rasterization engines typically compute the barycentric coordinates in screen space, which is why they need to do perspective correction and why texture sampling is harder. However the Möller–Trumbore ray-triangle intersection algorithm will return the barycentric coordinates on the real triangle as it exists in 3D space so we don't need to worry about perspective correction at all. Contrary to popular belief, ray tracing is simpler and faster than rasterization in many regards[6]. Now lets move onto harder stuff like transparency and shadows.

SEMI-TRANSPARENT SURFACES

To support transparency we will add a 4th stage in the rendering process, which will occur before the last shading stage. So to recap, we have the first stage where the 2D acceleration structure is computed, then the second stage where the acceleration structure is used to help us find the closest surfaces visible to the screen as fast as possible. Then in the third stage we check each ray to see if it intersected with a transparent or semi-transparent surface. If it did, then we continue checking for intersections until the ray hits an opaque surface.

So for every ray which first strikes a non-opaque surface, we create a list of all the surfaces the ray passed through until it hit an opaque surface. Then we do shading on each surface in the list and use alpha blending to combine the colors from each surface. Keep in mind that the order is important when blending the colors together because in real life when light passes through a semi-transparent material some wavelengths are absorbed[6]. Therefore final color will depend on the color and order of the surfaces the light passes through.

Again we can use linked lists to solve this problem because it's very similar to our other problem of sorting objects by distance. We must be able to insert new items at any point in the list very quickly but we don't need to access random items in the list, we just need to be able to iterate through it. We could use arrays with fixed sizes and then sort the items in the array after filling it with items, but that would be a waste of memory since most rays will usually hit an opaque surface first and we don't need a list at all for those rays.

The basic idea here is that for every ray we shoot into the scene, we build a list of surfaces the ray traveled through, so that we can see through surfaces which aren't totally opaque. That allows us to look through windows, or even several layered windows. The last item in the list will generally be the opaque surface which stopped the ray from traveling through it, but it would probably be a good idea to enforce a size limit on our list to prevent frame rate drops in areas containing many transparent surfaces visible through each other.

The process just described for rendering transparency is essentially the same as order-independent transparency techniques used by rasterization engines. The Wikipedia page for order-independent transparency suggests using linked lists to reduce memory overhead[7]: "Storing fragments in per-pixel linked lists provides tight packing of this data and in late 2011, driver improvements reduced the atomic operation contention overhead making the technique very competitive.", so it would appear linked lists are the right way to go.

LIGHTING AND SHADOWS

Once the lists are built we're ready for the last stage. To get the best results we will need to do shading on every surface in all our lists. Imagine a scene where you're looking out of a building window and into the window of another building. In order to get a correct picture of what's inside the other building, we need to know the correct colors of the surfaces behind both windows. Also, both windows will exhibit some degree of reflectivity and glare, so unless we do shading on every item in the list before blending them together, we wont get realistic results.

In order to do shading properly for each surface, we need to take into account all the lights which are shining light onto the surface. If we are shooting more than one ray through each pixel to reduce aliasing artifacts, and the scene contains many lights, then the shading stage can become quite demanding. Luckily there are also ways to optimize this process and make it much faster. One of those optimization methods comes from realizing that we don't actually need to check every light against every surface, only surfaces near the light.

The energy emitted by any source of light drops off exponentially with distance and so they usually only light up nearby surfaces. Once again we can employ the power of linked lists by creating a list of objects for each light. So for every light, we create a list of objects close enough to be affected by the light. We don't necessarily need to have the objects in each list ordered in any way, so a linked list may not be the best solution, but it still meets all our requirements by being memory efficient and fast if we just add new items onto the start or end of the list.

We would have to compute the object lists every frame but we could precompute lists for static lights and objects which we know will always remain in the same position. When it comes to the shading stage and we have a surface we need to shade, we loop through every light, checking whether the given surface is close enough to be lit up by the light. If it is close enough, then we know our object must be in the list for that light, but that's not particularly important. What's important is that we have a list of all the other objects close to the light.

To figure out whether or not the given surface is exposed to the light, we need to figure out whether any of the objects in the list are blocking the light. We create a ray which goes between the surface and the light then loop through every object in the list for that light, checking whether the ray intersects with the bounding sphere of each object. If the given object doesn't intersect with the ray then we know it cannot be blocking the light from reaching the surface and we can skip that object and any objects which aren't between the light and surface.

INFINITE RANGE LIGHTS

However not all lights in a video game scene should obey the inverse square law, some lights must have an infinite range and that presents some problems for us. If we want to create the effect of sun light, the light is going to have a huge range and so it becomes pointless to store a list of objects close to the light because it'll basically be a list of all objects in the scene. What we want to do is create a special class of light which is treated as having an infinite range so it can light surfaces at any distance.

Then we can position it extremely high in the sky like the real sun so that the scene receives what is essentially directional light. A light like that is going to shine on virtually all objects in the scene and create shadows far off into the distance. Not only that but objects outside the frustum are capable of casting shadows onto the visible portion of the scene. One obvious solution to this problem is to create a list which contains all the objects within a certain distance from the camera and use those as potential occluders for the light.

Since most scenes will probably contain only one or two lights with infinite range that isn't an entirely bad solution. It means that the sun will only create shadows on objects within a certain distance of the camera but most games do that anyway. There is another little tweak we can do though, which is to check whether or not each object is inside the frustum. If it is we add it to the occluder list of the light, but if it isn't then we may not have to add it to the list because it may not cast a shadow into the visible part of the scene.

To figure out whether or not its shadow would be visible to the camera, we can do a cone-frustum intersection test. The cone is created by placing the apex at the position of the light and then the body of the cone is positioned such that it perfectly contains the bounding sphere of the object. If that cone intersects with the viewing frustum then we know the object is probably casting a shadow onto a visible part of the scene so we add it to the occluder list. However if that tweak isn't used we only need one list of occluders for all infinite lights.

Since we're using the distance from the camera to decide which objects to add to the list, it's going to be the same for all lights with infinite range. However if we use the cone-frustum test to exclude some objects based on the position of the light then we will need a separate list for each light just like the normal lights. The objects inside the frustum could be stored separately because they will still be the same for all the lights with infinite range, but each light would still need its own list of objects which passed the cone-frustum test.

AREA LIGHTS & SOFT SHADOWS

The shadows created by point lights will have well-defined hard edges, but area lights will often produce soft-shadows where the edge of the shadow fades instead of suddenly stopping. The fading region of the shadow is called the penumbra and often occurs on real life shadows. When dealing with an area light we need to figure out how much of the area light is visible to the given surface. If a portion of the light is being occluded then the surface will only be partially lit, causing shadow penumbras. A simple way to do this is shoot several rays to sample the area light[5][8].

Now lets say we have a scene with many area lights, the numbers of computations required for shading a single pixel will grow rapidly as we increase the level of anti-aliasing because for every ray we shoot into the scene, we need to do shading on the surface it hits, meaning we need to send out multiple rays to each area light near the surface. If we're shooting multiple rays through every pixel to do anti-aliasing, we will end up sampling each area light a large number of times for a single pixel, and that's not acceptable if we want real-time rendering.

The following technique can be used to optimize shading with area lights but as we will see, it still requires a relatively large amount of computing resources to make the shadows look good. To simulate an area light with less computation we first break the area light up into several point lights, then for each ray sent through a single pixel, we only check it against one of those point lights. For example if we sent 4 rays through each pixel, then we would break each area light into 4 point lights, assigning only one point light to each of those rays.

This would give a fairly accurate approximation since we're still sending 4 rays to each area light to compute shading on every pixel. Most of the time when we send multiple rays through a single pixel into the scene, all the rays will hit the same surface and be fairly close to each other, which is why this method works. It would however be less accurate in cases where several of the rays we send through a single pixel hit different surfaces which are not close to each other. This can happen because the rays diverge as they travel into the scene.

LIMITATIONS OF REAL-TIME RENDERING

There is still one very serious problem with this approach though, which is that shadow penumbras will show banding artifacts[8]. Since the area lights are being broken up into a set of point lights the shadows for each light are distinct and it doesn't produce a realistic penumbra effect. The usual method for solving this problem is to apply random "jittering" to our shadow rays, which replaces the banding artifacts with noise artifacts[5], but in order to reduce the noise to acceptable levels requires that we use even more shadow rays.

The RNG used for ray jittering will also introduce more unwanted overhead, so overall this method is not ideal. In a moment we will see that there is a much faster method to handle ray traced area lights but before we look more at that, first it's worth mentioning some of the limitations of the ray tracing methods described so far. The 2D acceleration structure we're using will prevent elaborate effects such as refraction because if we change the direction of the ray mid-flight then the acceleration structure becomes useless.

For example, imagine a scene which has a glass cup near the edge of the screen, it's possible that a ray could hit the cup and then be refracted into an area of the scene which is outside the field of view, but the 2D acceleration structure only knows about objects within the field of view, and we want to keep it that way otherwise the rendering process would be much slower. Even if the ray didn't get refracted outside the scene, it would still be difficult to figure out which objects need to be checked due to the way we have organized the objects.

It would seem that if we want a real-time ray tracing engine, we need to accept many of the same limitations that come with rasterization engines. There's also the problem of reflections, if we do that with ray tracing it's extremely hard to optimize for real-time applications. Instead it's probably better to use efficient hacks such as environment mapping to produce less accurate reflections that are computed very quickly. It also turns out that another big problem for most rasterization engines is soft shadows / shadow penumbras.

ANALYTICAL SOFT SHADOWS

However some rasterization engines such as the Unreal Engine do have support for area lights and soft shadows. They achieve this by ray marching with distance fields[9], however there are a few unappealing aspects to that approach. The main issue is that the distance fields need to be precomputed, which means it will only work on static objects. Also, while it is quite a fast method as far as real-time soft shadows go, ray marching distance fields can still be a very intensive operation and can cause a substantial drop in the frame rate.

A more appealing solution would be something where we can analytically compute shadow penumbras. It turns out that this is possible because the penumbra size is based on a ratio between the distance to an occluder and the light. The formula is: *penumbra size = light radius * (occluder distance / light distance)*. Alexandru Voica published an article in 2015 describing how that formula can be used to compute soft shadows using only one shadow ray per pixel[10]. Using knowledge of the penumbra size we can use a blur kernel to blur shadow edges in screen space.

Since it's a ray tracing method it's also free of visual artifacts common to the other methods. It will work for dynamic objects and can also handle shadows caused by semi-transparent objects, so it's exactly what we need for a real-time ray tracing engine. When it comes to shadows and semi-transparent surfaces, another problem that we might also want to deal with is objects causing "colored shadows". For example when light shines through a stained glass window, the color of the light is altered by the color of the glass and produces a colored shadow.

COLORED SHADOWS

Rasterization engines usually only support non-colored shadows, even semi-transparent objects will produce a normal shadow as if it was completely opaque, like any other object. But that's obviously not what would happen in reality, because semi-transparent objects would allow some light to get through, and therefore it wouldn't produce such dark shadows. Not only that, the light which does get through the object may be tinted a specific color and cause a "colored shadow", and unsurprisingly they are a bit more demanding to generate than normal shadows.

When doing normal shadows, we can stop checking objects against our shadow rays as soon as we find a single opaque surface intersecting with the ray. If we want to produce colored shadows then we also have to check whether or not each intersecting surface will allow light to pass through it and if it does then we must add the color of the surface to yet another list so that we can calculate the final color of the light after passing through any semi-transparent surfaces. Again, linked lists are a potential solution to this problem.

If the ray does hit an opaque surface along the way then the list can simply be discarded and we can apply normal shadowing to the given surface because we know the light cannot reach it at all. If nothing is blocking the shadow ray from reaching the given light then we can use the list of colors just created to figure out what color the light source is contributing to the given surface after the light has passed through one or more semi-transparent surfaces. However it seems very unappealing to use even more linked lists if we already use so many.

Lets keep in mind that people don't closely examine shadow colors and it's rare for light to travel through multiple semi-transparent surfaces. A more optimized but less accurate solution to this problem is to simply average the colors of the semi-transparent surfaces and not worry about their order. The resulting color is then blended with the color of the light to get the shadow color. Since semi-transparent surfaces will block some of the light, the light intensity is also scaled according to the cumulative alpha value of all semi-transparent surfaces.

CONCLUSION

In summary, we have examined several ray tracing optimization techniques which can be combined to produce what should theoretically be capable of achieving real-time rendering speeds. We wont really know for sure until these algorithms are actually implemented to run in a highly parallel fashion on a GPU device. A prototype implementation has been written to run on the CPU and the source code is available on GitHub[11] but it's not yet very fast since it's mainly written for readability and doesn't use SSE, multi-threading, or anything else to speed it up.

While the rendering process just described does have many of the same limitations that rasterization engines are faced with, it also avoids many of the pitfalls that come with the rasterization process and it gains several of the advantages that come with ray tracing, such as improved shadows and transparency. Having said that, a hybrid rendering system which uses rasterized deferred rendering in conjunction with ray traced shadows and lighting, may very well turn out to be the best real-time solution because it gives us the best of both worlds.

The concepts discussed in this paper have been simplified, if you want a deeper understanding of how some of the algorithms discussed work, then it's highly recommended to read the referenced source code. Some important concepts such as vertex caching have been left out of this paper but you will see it implemented in the code. The subject of 3D rendering goes very deep, we haven't even examined topics such as volumetric lighting, subsurface scattering, ambient occlusion, or physically based rendering, but most of them can be solved with ray tracing methods.

REFERENCES

[01] www.euclideon.com/technology-2/

[02] www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_7_Kd-Trees_and_More_Speed.shtml

[03] www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/

[04] www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection

[05] www.cs.cmu.edu/afs/cs/academic/class/15462-s09/www/lec/13/lec13.pdf

[06] www.gamasutra.com/blogs/AlexandruVoica/20140318/213148/Practical_techniques_for_ray_tracing_in_games.php

[07] www.en.wikipedia.org/wiki/Order-independent_transparency

[08] www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_5_Soft_Shadows.shtml

[09] www.docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/RayTracedDistanceFieldShadowing/index.html

[10] www.embedded-computing.com/articles/ray-tracing-for-beginners/

[11] www.github.com/JacobBruce/Ray-Tracer-CPU-Prototype