Hi everyone, my name is Michal Iwanicki.
I’m here today with Peter-Pike Sloan and I’m going to talk about a family of new spherical basis function that we call Ambient Dice (AD).
Something that I really want to cover today is not only the details of the research itself, but also what were the motivations for certain choices that we made, what were the constraints on our solution.

Because we work in games, we work for Activision Central Tech, doing R&D for various studios, most notably for the Call Of Duty franchise.

And even though it is “R&D” it’s mostly driven by the “D” part – which means that we’re doing the research for a particular title, it needs to be ready on time, we are actually responsible for putting it in the final game, and it needs to run within a very strict performance and memory constraints.
Most recently we’ve been working on lighting, so I will give you a brief introduction to lighting in games.

Traditionally, game lighting is separated into two parts that are handled separately. First, there is direct lighting, with contributions from traditional point, spot, and recently area lights, which are often shadowed with some form of shadowmaps. Then there is the indirect lighting, which is different in that it usually involves some sort of precomputations, storing the results in some data structures, and then resampling the results during the actual rendering. It’s generally computed at a spatial frequency that’s unrelated to the actual final rendered image.

The indirect lighting data usually comes in two flavors: there are hemispherical signals that describe lighting on surfaces, and fully spherical signals for describing lighting in space.

We often use different methods for representing the two.
We’ll focus on indirect lighting here. To be even more precise, we will focus on the diffuse contribution and how it is stored and represented, and we want it to represent a full spherical signal.

There are a number of solutions used in games.
I don’t really think we can say that this is actually a solved problem.

The solutions range from constant ambient which looks bad, to a combination of ambient and diffuse light.

The latter does a reasonable job when used to represent hemispherical signal, but fails miserably when representing spherical signal, generating those pee-stain like artifacts when the direction of the light changes rapidly.

Then there are low order spherical harmonics (SH): second and third order.

The problem with the second is that the quality is not really enough, and the third order is slightly too expensive.

Recently, people have also been using sets of spherical Gaussian (SG) lobes, which are fairly expensive, but have the advantage in that they can do a reasonable job representing radiance for low gloss specular materials.

And then there’s Ambient Cube (AC), which is a set of 6 clamped cos^2 lobes oriented along cardinal axes.

The quality they offer is fairly low, but it is really cheap.
INDIRECT LIGHTING IN GAMES

• Constant ambient
• Ambient + directional light [Car99]
INDIRECT LIGHTING IN GAMES

- Constant ambient
- Ambient + directional light [Car99]
- Low order spherical harmonics (2nd/3rd) [Ram01][PPS08]
INDIRECT LIGHTING IN GAMES

- Constant ambient
- Ambient + directional light [Car99]
- Low order spherical harmonics (2nd/3rd) [Ram01][PPS08]
- Spherical Gaussians [Tsai06][Wang07][RAD15]
INDIRECT LIGHTING IN GAMES

• Constant ambient
• Ambient + directional light [Car99]
• Low order spherical harmonics (2nd/3rd) [Ram01][PPS08]
• Spherical Gaussians [Tsai06][Wang07][RAD15]
• Ambient Cube [McTag04] [Hook16]
We need something that is really, really cheap. Most of our games need to run at 60Hz, which means we have ~16.6 ms to render a whole frame. This gives us around 7-8ms to do the actual scene rendering. The rest is spent on shadows, special effects, etc. At the same time, we don’t want to compromise quality. For instance, we want the lighting on objects to change spatially. But since it’s only indirect lighting, we don’t really need it to change at very high rate. A single sample every meter is roughly the density that we are aiming for. This means that when rendering objects we will need to use some sort of volumetric structure to store the lighting. Neighboring pixels will use mostly the same pieces of that structure. They will be accessing lighting data from roughly the same areas of memory. The traffic between memory and L2 cache will be relatively small. But on the other hand, it *will* be different for every pixel. Every pixel *will* need to use slightly different data, and one of the main goals for us was to limit that amount of data.
Why? Because accessing memory from a shader is really costly in terms of the resources it needs.
The diagram here shows a simplified view of a compute unit on both of the current generation consoles.
To access memory we need to allocate some registers to store the results.
It needs to send the actual fetch request to the texture unit, where it takes some amount of space in the internal queues.
Then there is the bandwidth utilization between the compute unit and L2.
Our main problem is that even without doing anything, just by using constant ambient, all the other features that we have in the shader cause us to be limited by the register counts that we use.
We’re almost limited by the queues in the texture units, and while we have some slack in the available bandwidth, we really want to avoid adding too many memory access requests.
All these factors are in a delicate balance: you can use more registers, but then you limit the number of shader instances you can have in flight.
This, in turn, limits their ability to hide the memory lookup latency, etc.
16KB of L1 / ( (avg) 4 wavefronts/SIMD * 4 SIMDs/CU * (avg) 4 lookups in flight * 64 bytes/cache line = 4096bytes ) = 4, so there’s room to hold 4 cache-lines per
wavefront per lookup.
There are 64 threads in each wavefront, each thread performing lookups from independent positions in textures.
Lookup positions are usually close within a wavefront, so it might not require all four cache lines, but there’s also not really a lot of temporal reuse.
Mostly there is spatial reuse, with the L1<->L2 path constantly busy.
OUR SETUP

- Texture read requires:
  - VGPRs to store the results
  - Space for addresses/commands in the texture unit queues
OUR SETUP

- Texture read requires:
  - VGPRs to store the results
  - Space for addresses/commands in the texture unit queues
  - CU <-> L2 memory bandwidth
- Already limited by those factors, in that order
The table here shows the number of coefficients that have to be fetched from the memory to compute the diffuse lighting in some of the cases discussed before. The interesting thing here is the AC basis, which doesn’t need all the coefficients to perform the evaluation. This is due to the fact that it uses the clamped cos^2 lobes, which have a local support.

This is a great property, and our main motivation was to find a modern alternative for that, something that would also be locally supported but provide better quality, comparable to at least 3rd-order SH (but at a lower memory access cost). We were fine with using more memory total as those are things we can reason about and solve separately, as long as we wouldn’t need to access it all.
AC is based, unsurprisingly, on a cube: each lobe points to one of its faces. A cube has the convenient property that it’s really easy to figure out which 3 faces are aligned with a given direction, and thus which 3 lobes we are going to need. We just take the signs of the direction vector coordinates. Our idea was to find a similar indexing scheme for some other regular polyhedron. After looking into it a bit, we found one for icosahedron, and we based our basis on that.

The basis comes in different variants, and we called the whole family Ambient Dice, as a reference to Ambient Cube and 20-sided die from pen and paper role-playing games.
The general idea is to take the icosahedron, find an efficient indexing scheme and create a basis through scattered data interpolation.

We would store something at the vertices and/or faces of the polyhedron, and define some way of interpolating those values across the whole sphere, and this will give us the spherical basis.
First, the indexing scheme.
If you look at an icosahedron it’s not really obvious how to figure out which triangle you’re pointing to.
We base the indexing scheme on the construction of the icosahedron from an octahedron.
If we take the octahedron oriented along cardinal axes, and split its vertices into two, insert extra edges there, and expand them when all the edges are the same length, we get an icosahedron.
ICOSAHEDRON

• Construct it out of octahedron
ICOSAHEDRON

• Construct it out of octahedron
ICOSAHEDRON

- Construct it out of octahedron
It has some convenient properties.
First, all its vertices are of one of three forms, where capital phi is the reciprocal of the golden ratio.
As you can see they are just cyclic permutations of each other, there are 4 vertices in each of such groups, two antipodal pairs, and we can assign an index to them by extracting the sign bits: group A will have indices 0-3, group B will have 4-7, etc.
ICOSAHEDRON

• 12 vertices, 4 of each of the 3 forms:
  • A: (±1, ±Φ, 0)
  • B: (0, ±1, ±Φ)
ICOSAHEDRON

- 12 vertices, 4 of each of the 3 forms:
  - A: \((\pm 1, \pm \Phi, 0)\)
  - B: \((0, \pm 1, \pm \Phi)\)
  - C: \((\pm \Phi, 0, \pm 1)\)
An icosahedron has 20 faces, all triangles, and we divide them into “base” triangles, that are entirely contained within an octant of a coordinate system, and the 3 groups of “side” triangles, where triangles in each of group intersect one plane of the coordinate system.
When we want to figure out which triangle a given direction points to, we first figure out which octant it’s pointing to, by simply extracting the sign bits from the direction vector.

Given the octant there are 4 triangles to choose from and you can make the choice with some simple math. In the end you have an index of a triangle, the coordinates of its three vertices, and their indices from 0 to 11.

The pseudo-code is in the paper, and it’s actually super simple.

Given the octant we can either point to the base triangle or one of the side triangles, each belonging to a different group.

We can just check the direction against the planes passing through the edges of the base triangle.

Because of all the symmetries, this can be done in the +++ octant, and the dot products needed for the tests are computed against vectors with one coordinate equal to 1, so they boil down to two multiply-adds.

Each vertex of the base triangle is from a different group, so if we point to the base triangle, we just use them, multiplied by the bit signs from the direction.

If we use any of the side triangles, one of the vertices gets replaced with a vertex from another group with one coordinate flipped.

And however complicated this sounds, the code for that is actually really simple, and
it’s in the paper.
Now we need to figure out how to interpolate the data across the icosahedron. Of course the most obvious solution would be to just use linear interpolation: that would only require storing a single RGB value for every vertex, and evaluation would only require fetching the 3 values of the vertices of the triangle we’re pointing to. Unfortunately, this looks pretty bad. The mach bands are really apparent, and even using more fancy spherical linear interpolation doesn’t really help either.
We tried a lot of different options, and the thing that worked really well were hybrid Bezier patches. Hybrid Bezier Patch is just a convex combination of three quadratic bezier patches that have a different center control point. To construct the patch we need to store signal and its directional derivatives on the vertices of the icosahedron. This uniquely identifies the 9 control points of the patch, and then the remaining one is chosen to create C1 continuity across neighboring patches. This can be done without any extra data if we make the signal change linearly instead of quadratically along the shared edge. The reconstruction consists of finding the triangle, grabbing the vertex data, computing the control points and then evaluating the hybrid Bezier patch. And this looks really great, the quality of the reconstruction of the diffuse irradiance signal is comparable to often even 5th order SH.
However, it was a bit too costly for our use-case, because it requires a fair share of ALU and we need 27 coefficients for the reconstruction. That’s the same as for 3rd order SH, and even though the quality is higher, we would happily trade some of it for cheaper reconstruction. Our first solution for that we was to switch to YCoCg color space and use the higher-order interpolation only for the luminosity component, as this is what our eyes are most sensitive to, and use simpler, linear interpolation for the chromaticity components. And while it does generate some minor discolorations, they are very localized, contained within single faces of the icosahedron. In general, there’s almost no quality loss. And it brings the cost down quite a bit – because we now need way less data for the reconstruction: only 15 coefficients.
RADIAL BASIS FUNCTIONS

• Still a bit too costly for us (ALU)
• No luck with RBFs restricted to single triangles
• BUT! Clamped $\cos^2$ (scaled by $1/2$) forms partition of unity
• $\cos^4$ (scaled by $5/6$) as well
• Use 70%-30% combination of the two
• 18 coefficients, but super simple ALU

However, we still weren’t entirely happy, it was still a bit too costly in term of ALU. While investigating different radial basis functions (RBF) we noticed that clamped $\cos^2$ forms partition of unity over a sphere, and so does $\cos^4$. We tested different combinations of the two and settled on 70%-30% mix of those two lobes.

For evaluation we need 6 lobes, intersecting the hemisphere pointed to be the evaluation direction, so 18 coefficients for reconstruction.

In return, we get a much simpler reconstruction logic.
VISUAL COMPARISON

IMPORTANCE SAMPLING
VISUAL COMPARISON

AMBIENT CUBE
VISUAL COMPARISON

SPHERICAL GAUSSIANS (12 LOBES)
VISUAL COMPARISON

AMBIENT DICE YCoCg
VISUAL COMPARISON

AMBIENT DICE RBF
VISUAL COMPARISON

IMPORTANCE SAMPLING
The paper covers some interesting details on the how to perform projection of the signals onto the basis, including cross-projection from different bases, but I wanted to mention here us one of the metrics that we use for the evaluation of the quality of the reconstruction.

We compute how well the given bases represents spherical harmonics of a given order. We can construct the matrix that takes the SH signal, projects it onto a given bases and then back to SH. In an ideal situation this would be an identity matrix, but it generally isn’t, and we can compute the Frobenius norm of the difference. This gives us a single number describing how well the bases represents a given order of SH.

AD in the Bezier patch versions works really well up to 4th order SH, and reasonably at 5th order.

AD in the RBF setting works well up to 3rd order SH.

<table>
<thead>
<tr>
<th>Method</th>
<th>SH2</th>
<th>SH3</th>
<th>SH4</th>
<th>SH5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambient Cube</td>
<td>0.10825</td>
<td>1.73543</td>
<td>3.16228</td>
<td>4.3589</td>
</tr>
<tr>
<td>Icosahedron linear</td>
<td>3.543e-3</td>
<td>0.18454</td>
<td>2.01751</td>
<td>4.13203</td>
</tr>
<tr>
<td>Icosahedron cos²</td>
<td>2.229e-3</td>
<td>3.153e-3</td>
<td>2.0001</td>
<td>4.12316</td>
</tr>
<tr>
<td>Ambient Dice RBF</td>
<td>3.971e-4</td>
<td>1.427e-2</td>
<td>2.0001</td>
<td>4.12316</td>
</tr>
<tr>
<td>Ambient Dice RGB</td>
<td>3.857e-4</td>
<td>1.539e-3</td>
<td>2.777e-2</td>
<td>0.17101</td>
</tr>
</tbody>
</table>
We also have data on the performance of the methods implemented in the Call Of Duty engine. The timings are just for the rendering of the opaque pass, and we use two methods there for the indirect lighting – larger structural meshes use lightmaps and everything else uses the local, per-object volumes that store the indirect lighting in various representations. I think the important bits from that table are the rows marked in red - SH3 and AD with RBF – the Ambient Dice is around 0.2 ms faster, placing AD between 2nd and 3rd order SH in terms of cost, while getting the quality of 3rd order SH. The other bit is the YCoCg version. That’s more expensive due to the more complicated ALU, but comparable to using 12 spherical Gaussian lobes. The RBG version is of course very expensive both because of multiple texture fetches, and complicated ALU.
And here is the earlier table with the new rows. While the total number of coefficients is higher with AD, they are well behaved coefficients, just colors and their derivatives, so it’s easy to reason about them in terms of compression. Especially the RBF variants, which need just the colors, and can be efficiently compressed with modern block compression.

<table>
<thead>
<tr>
<th>Method</th>
<th>Coefficients total (per color channel)</th>
<th>Coefficients for evaluation (per color channel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd Order SH</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3rd Order SH</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Spherical Gaussians (12 lobes)</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Ambient Cube</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Ambient Dice RGB</td>
<td>36</td>
<td>9</td>
</tr>
<tr>
<td>Ambient Dice YCoCg</td>
<td>20 (average, 60 for full RGB)</td>
<td>5 (average, 15 for full RGB)</td>
</tr>
<tr>
<td>Ambient Dice RBF</td>
<td>12</td>
<td>6</td>
</tr>
</tbody>
</table>
Subdivided octahedron addressing is simple, see octahedral env mapping/normal encoding. However it needs to duplicate certain values. Any logic to remove the duplication blows up the ALU cost, way above the cost of icosahedron addressing. Encoding normal vectors works, although the decoding cost is twice the cost of the octahedron normal encoding. The resulting quality of the reconstruction is higher, but the extra quality might not be required by many applications.
QUESTIONS

• Acknowledgments:
  • Derek Nowrouzezahrai, Josiah Manson and Bart Wronski
  • Yuriy O’Donnell and David Neubelt – Probulator [Prob]
  • Infinity Ward and Sledgehammer Games
  • Jennifer Velazquez and Kris Magpantay
REFERENCES

- [Car99] John Carmack: Quake 3 Engine
- [Ram01] Ravi Ramamoorthi, Pat Hanrahan: An Efficient Representation for Irradiance Environment Maps, Siggraph 2001
- [PPS08] Peter-Pike Sloan: Stupid Spherical Harmonics (SH) Tricks, Game Developers Conference 2008
- [RAD15] David Neubelt, Matt Pettineo: Advanced Lighting R&D at Ready At Dawn Studios, Siggraph 2015, Advances in Real-Time Rendering course
- [Tsai06] Tsai Y.-T., Shih Z.-C.: All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation, ACM TOG 2006
- [Prob] https://github.com/kayru/Probulator