
SIGGRAPH2011

Voxels in LittleBigPlanet 2

Alex Evans & Anton Kirczenow

Advances in Real-Time Rendering in Games

Boutique engine coding

- LBP, like LBP2, uses a custom engine, tailored to the game's look and design.
- What's the biggest constraint?
  - GPU Time? Sorta.
  - CPU (PPU) Time? Ish.
  - Programmer Time! Definitely.
    - LBP1 - 1.5 graphics coders x 36 months
    - LBP2 - 1.1 graphics programmers x 18 months

about 10% is R&D you throw away; 10% is R&D you use; 30% is implementation; 50% is bug fixing, bullet proofing and optimising (time & memory)

time was the greatest constraint. the rule was: if it was good enough, move on to the next problem. in many cases in this talk you'll probably perceive branch points in possible implementations; one take away from this talk is how often we got away with 'good enough'.

Coder time as constraint

- Typically your time breaks down as:
  - 15% R&D you throw away
  - 5% R&D you use
  - 30% implementation time
  - 50% bug fixing, optimizing, ‘bullet proofing’

Wednesday, 10 August 11

about 10% is R&D you throw away; 10% is R&D you use; 30% is implementation; 50% is bug fixing, bullet proofing and optimising (time & memory)

time was the greatest constraint. the rule was: if it was good enough, move on to the next problem. in many cases in this talk you'll probably perceive branch points in possible implementations; one take away from this talk is how often we got away with 'good enough'.

LBP Engine requirements

- many local lights please
- simple code
- predictable cost:
- μ_σ ***much*** better than $\mu\sigma$

Lighting a UGC Game

- Goal: fast evaluation of lighting from multiple moving light sources in a dynamic scene
  - Constraint 1: Scene has low depth complexity - '2.5D'
  - Constraint 2: Has to run at consistent speed with modest memory usage, and no pre-computation

So we tried a few things...

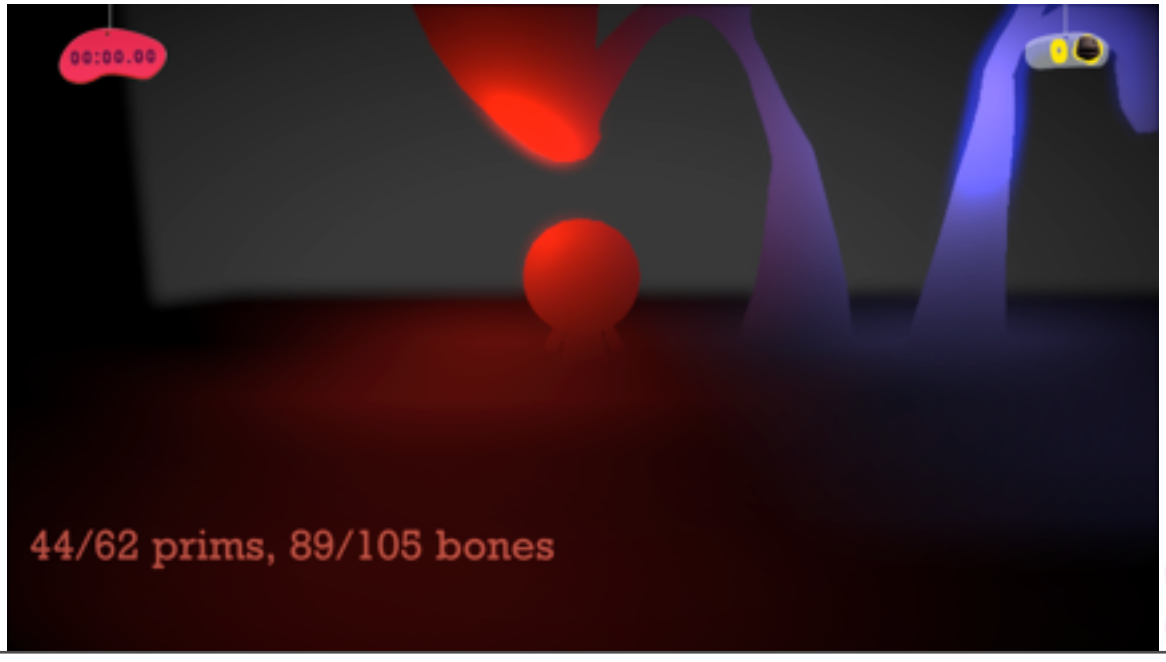
- LBP0 aka SIGGRAPH 2006
  - Irradiance volumes
    - sliced in screen-space
- LBP1 aka SIGGRAPH 2009
  - Ilight pre-pass renderer
    - with 2 layers of depth for transparency
- LBP2 aka SIGGRAPH 2011
  - dynamically voxelized scene
  - world space irradiance volume

Attempt 0: View aligned Irradiance Volumes

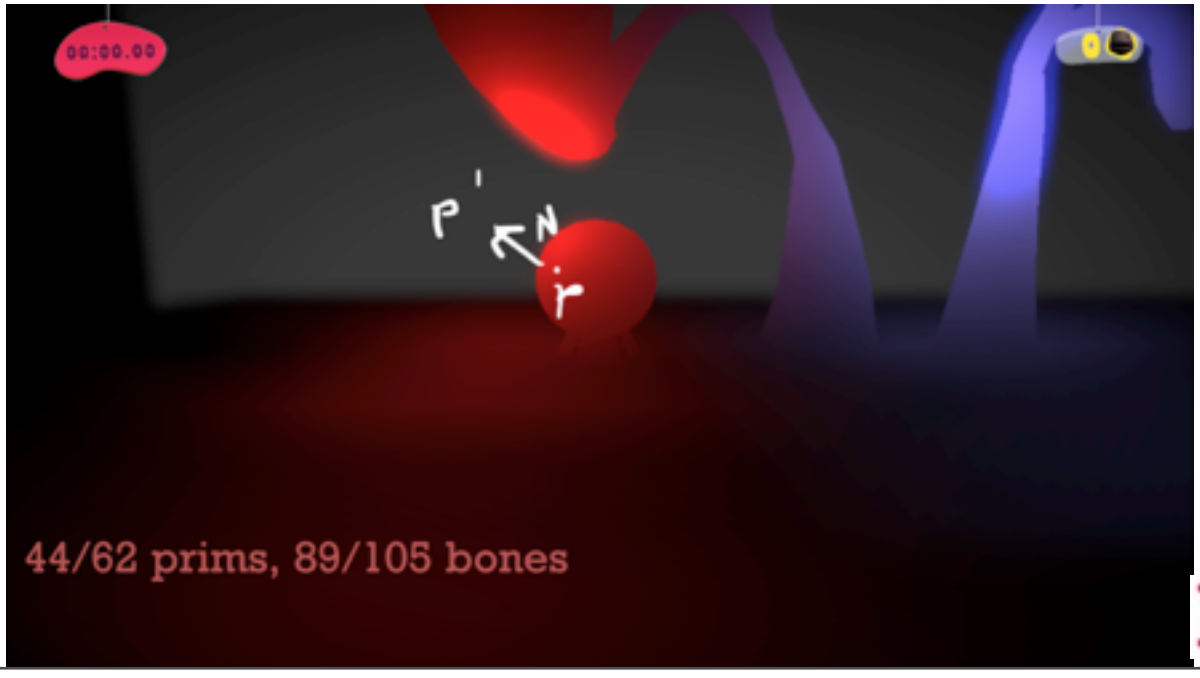
- In a dynamic scene, it's not possible to precompute the irradiance volume
  - So we are going to recompute it on the fly using the GPU, at low resolution, based on a potentially large number of light emitters
- Since we're recomputing it every frame,
  - It makes sense to compute it in screen space.
  - In this example, the target constraint was for a '2.5D' thin world, so a small number (16) of slices are used, parallel to the screen. They are evenly spaced in post projective space, ie evenly spaced in 'w'. (1/z)

The scene lit by a single point source

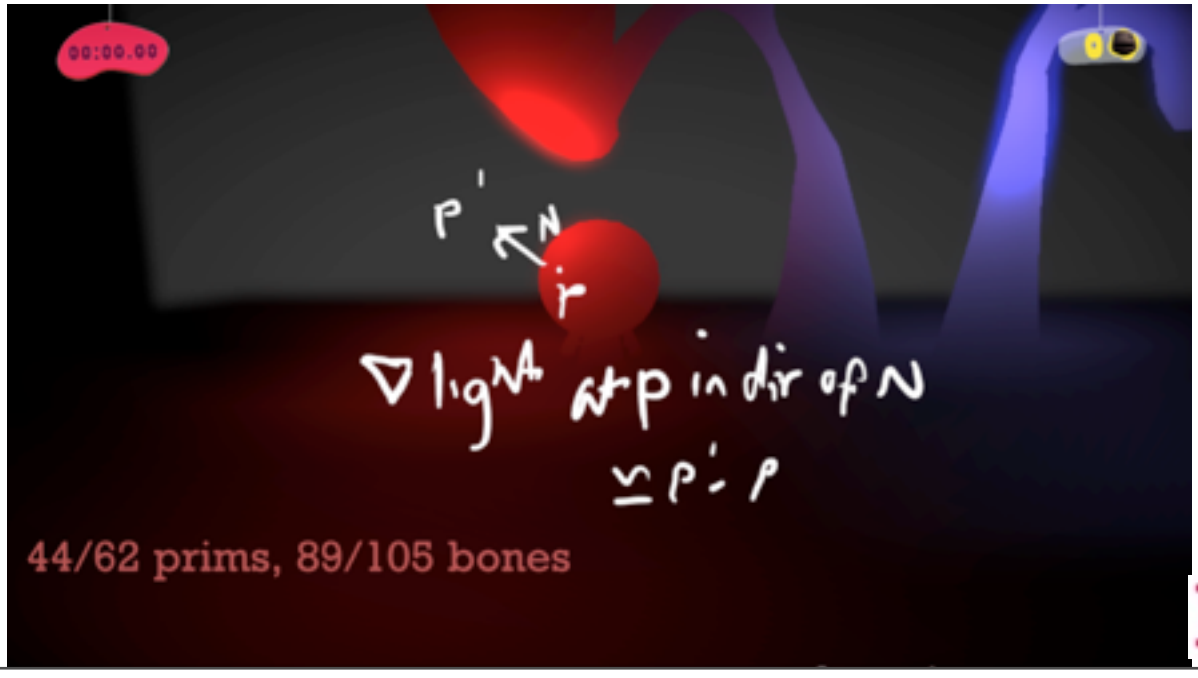
Irradiance volume visualisation



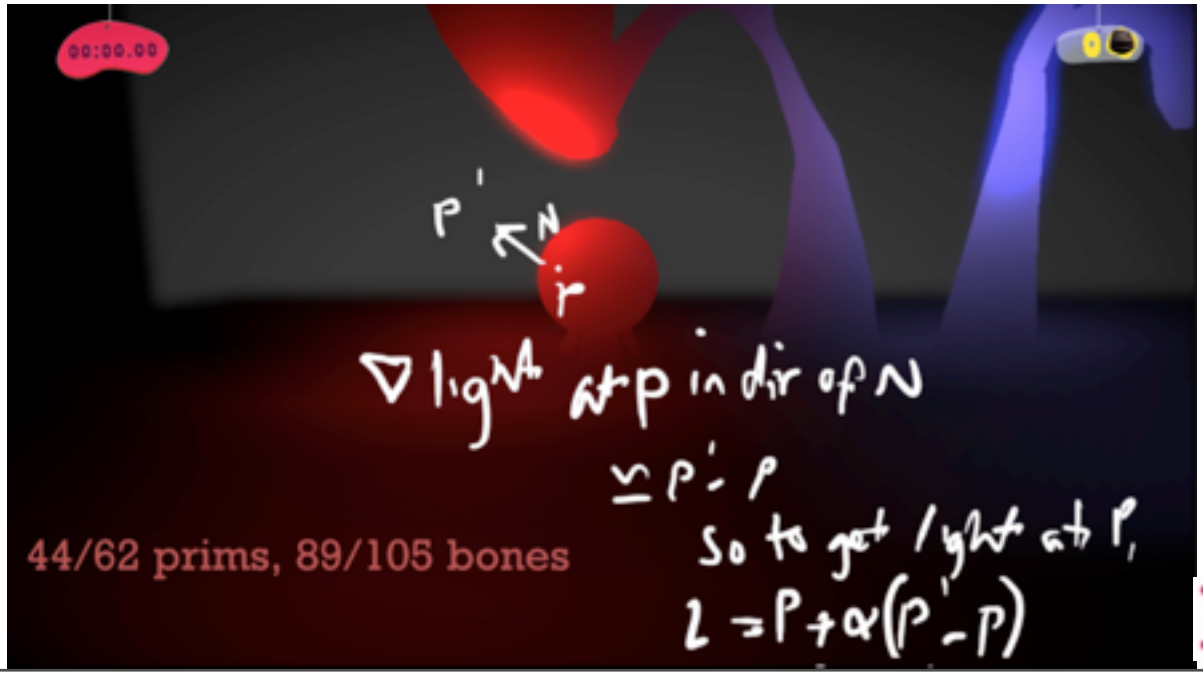
Using ∇ to approximate directional dependence



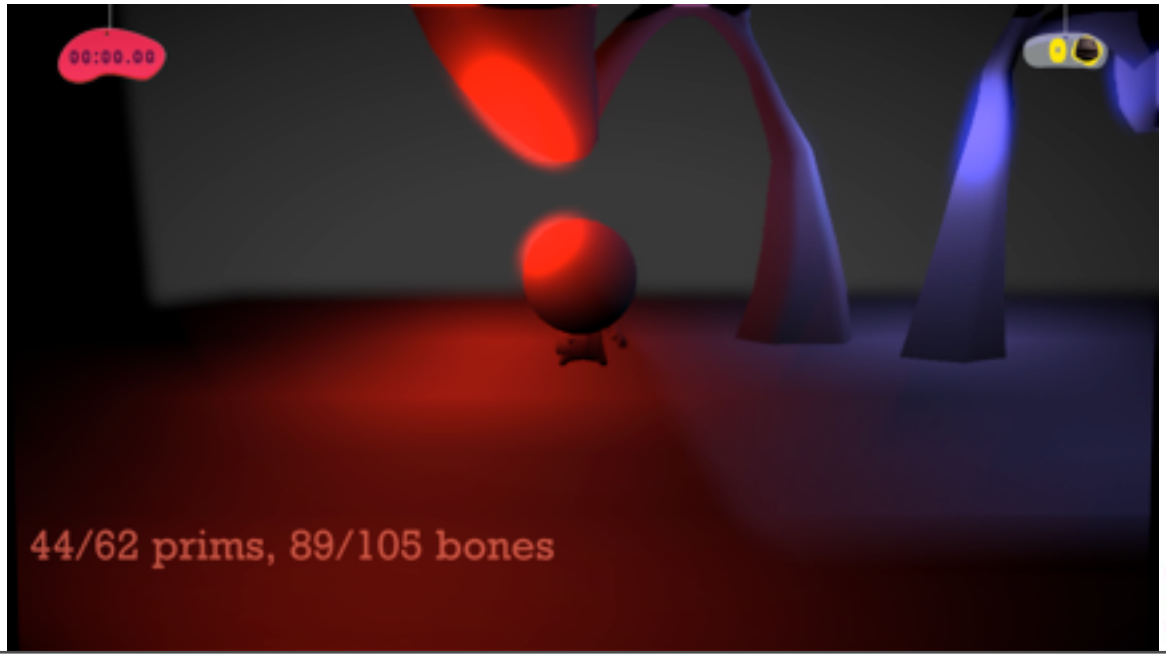
Using ∇ to approximate directional dependence



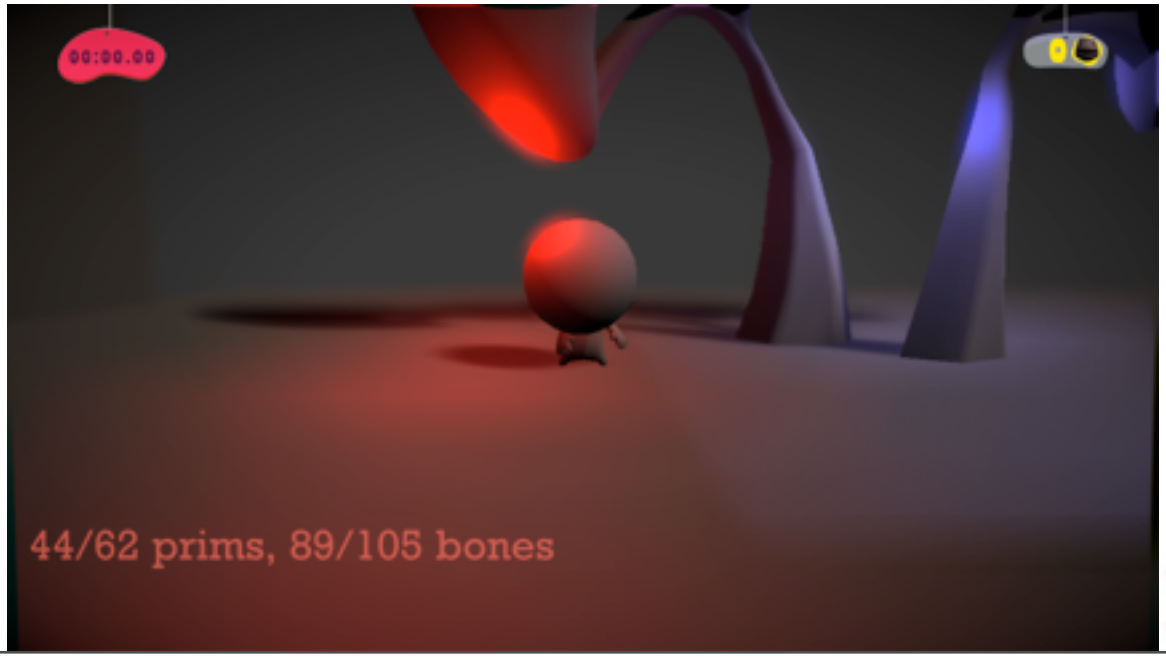
Using ∇ to approximate directional dependence



Results (no sun)

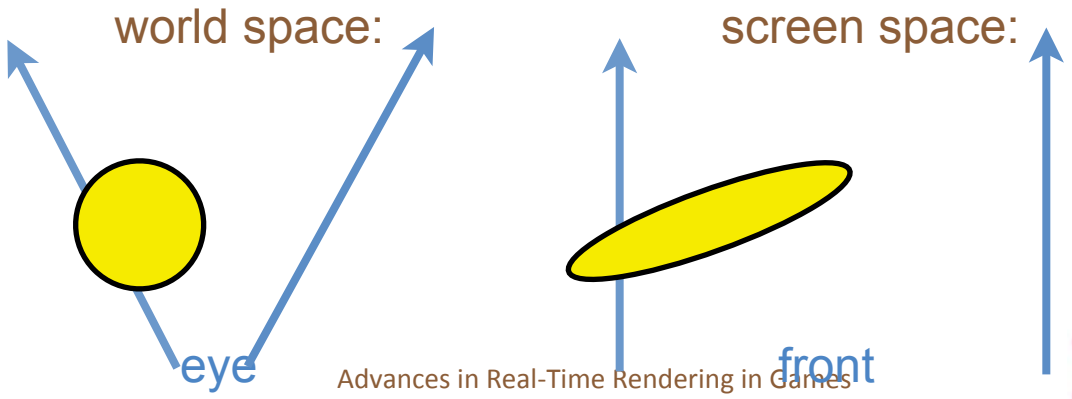


Results (with sun)



Why that did not work

- Light leaking (no occlusion)
- Low slicing rate + high perspective



Wednesday, 10 August 11

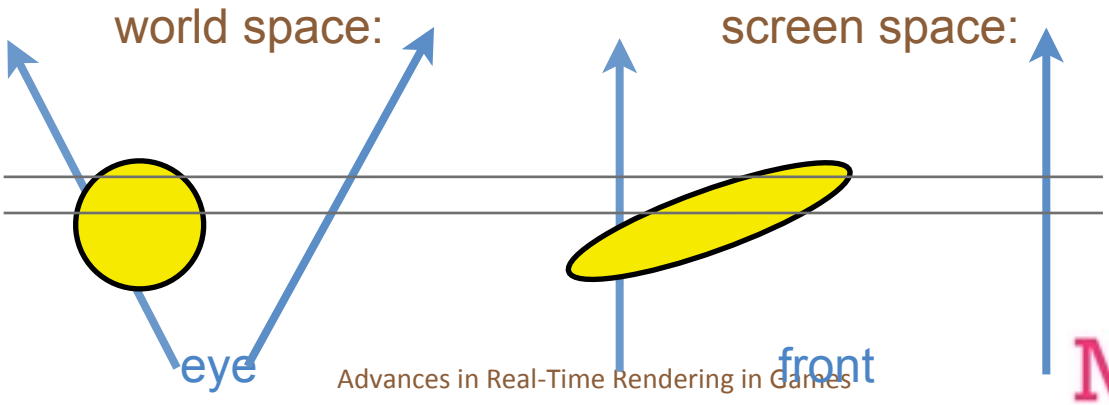
LBP0: Irradiance volumes [siggraph 2006]

greatest single problem: low rate of z slicing in post-perspective space -> eccentricity of ellipsoids high enough that the slices shear apart.

also, light leaking; lack of occlusion.

Why that did not work

- Light leaking (no occlusion)
- Low slicing rate + high perspective



Wednesday, 10 August 11

LBP0: Irradiance volumes [siggraph 2006]

greatest single problem: low rate of z slicing in post-perspective space -> eccentricity of ellipsoids high enough that the slices shear apart.

also, light leaking; lack of occlusion.

Attempt 1: aka how we actually did it in LBP1

- Light pre-pass renderer
  - lighting run at 2xMSAA *sample* rate
  - allows for 2 layers of transparency!!!!
  - all lighting is simple deferred, unshadowed.

Wednesday, 10 August 11

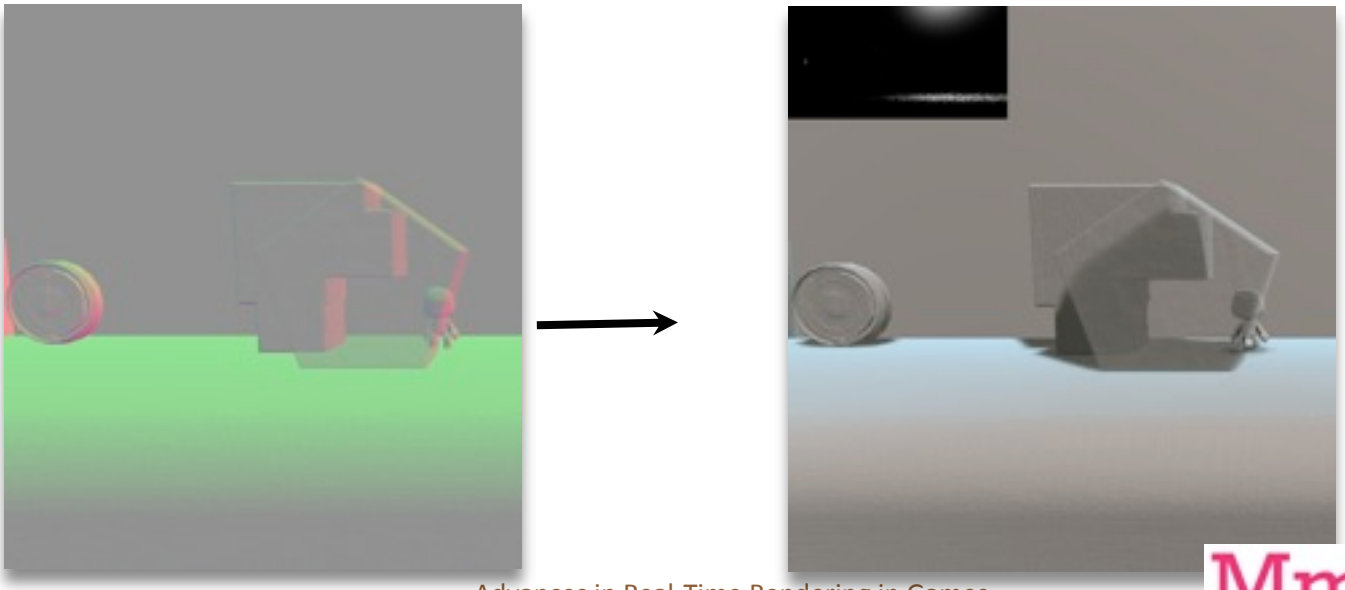
LBP1: Light pre-pass rendering [siggraph 2009]

greatest single problem: 2 < many

Light pre-pass renderer

SIGGRAPH2011

Lay down a Z/N pre-pass at 2X MSAA



Mm

Wednesday, 10 August 11

The solution we shipped with amounts to (I discovered yesterday researching the other talks at this course!) Wolfgang Engel's light-prepass algorithm ,or a variant of it. So we're in good company :) and I won't cover that aspect of it any more today as I'm guessing Wolfgang will have covered it and many extensions already.

For LBP the technique evolved less from a 'prepass' mentality, and more through my self imposed allergy to MRTs, combined with the observation that every single one of the materials generated by the artists to that date had the same specular power. (!) (which happened to be the default i'd set - 22. luckily I'd spent some time choosing that value - but it's worth noting: whenever you choose a default for your artists, do so carefully, and visually. there is never any excuse for hiding behind the old excuse of 'oh its just coder art some artist can make it look good later' if you're a graphics coder)

I figured that I might as well light the whole scene with one 'grey plastic' shader, whose material properties were constant and thus didn't require any screen space material buffers, and then 'paint the color' over the scene in a 2nd geometry pass. The specular lobe was broken out into the alpha channel so that it could be modulated separately from the diffuse component.

This had the added benefit that it forced me to revisit both 'spite lights', but also MSAA and transparency, both of which were now 'weaker' (aka not working) than they had been in the previous forward renderer.

Light pre-pass renderer

then re-render scene ‘forwards’, sampling ‘L’ into BRDF

a brief note on performance: for typical sunlit scenes, the new 2-pass system was relatively well suited to the peculiarities of the RSX GPU: the first pass sampled only normal maps, and acted almost like a fast z prepass (although I couldn’t use the double speed rendering mode, it’s still a relatively small amount of the total frame time, at under 3ms). This helped reduce overdraw for the 2nd pass with its extremely bandwidth-hungry multi-layered textures, and reduced thrashing of the texture cache: the normal maps (which were themselves often layered) were ‘out the way’, balanced between the two passes. So the performance penalty of rendering the scene geometry twice was largely offset.

2 layer transparency

SIGGRAPH2011

deferred shading with alpha. what?

Mm

Wednesday, 10 August 11

the new deferred renderer was fine, but immediately threw into question how to light transparent objects. LBP very often has semi transparent objects placed over the scene, most obviously in 'create mode' where a 50% alphad 'stamp' object can be positioned in the scene before being 'materialised' by hitting X.

So it was extremely important to be able to light scenes including transparent objects. I was unwilling to introduce a second code path for those objects.

In addition, transparent rendering wasn't the only problem: any technique that reads back the z buffer has issues with transparency.

as I mentioned at the outset, the visual style of LBP was 'miniature world' and heavy DOF & motion blur were the most important techniques to achieve that 'real world mini' look

these techniques rely on post-processing of the rendered scene, typically taking into account the Z buffer value at each pixel. Thus, they break down on scenes with transparency.

In the switch to a deferred shader, the main surface renderer also now fell into this 'transparency is hard' category.

LBP uses 2X MSAA, and it occurred to me that it would be nice to trade spatial antialiasing for the majority of the screen, with layered transparency where it was needed. In other words, use the memory normally reserved for related color samples in a pixel, to store 2 layers of both Z and N, but only where it was needed. By running the deferred lighting passes at double horizontal resolution, and by making the post-processing effects aware of this 'dual layer transparency', it was possible to create correct lighting, DOF and motion blur for the very-common-in-LBP case of a single semi-transparent object in front of solid objects.

but: 2 is not a big number

leads to odd effects when you get more than 2 layers:

users actively are exploiting this to great effect!

Obviously, this 2 layer system wasn't without its costs.

It led to relatively 'fragile' shaders which carefully offset their input UV coordinates by half a pixel here and there - not intuitive looking code, and a hassle when supporting multiple resolutions or off-screen rendering.

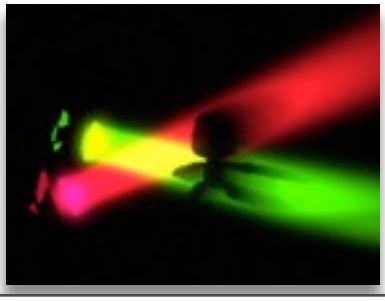
worst of all, you run out of '2' very quickly! glass-behind-glass disappears. Amazingly, the community now uses this (along with another bug in the handling of fog!) to striking effect: quite a few levels place a thin layer of glass across the front of the entire level. now, all objects made of glass in the main play area disappear, since the alpha layer is entirely used up by the transparent foreground layer. Now, these entreprising creators can create 'magical' levels where the mechanism is hidden, or objects appear to float. In the hands of the right users, no mis-feature is ever all bad....

Worse, it slowed down some shaders - especially the final MSAA resolve shader, and the DOF shaders - because they had to take great care to treat the two samples separately.

Since LBP was released, Matt Swoboda and the Phyre Engine team have done some very interesting work on using the SPU to do deferred shading. In particular, their optimization of categorizing tiles of framebuffer according to need - eg transparency, no transparency- could be used to both speed up the easy cases, and further improve the quality/features of the 'deferred transparency' (perhaps with more layers). A particularly interesting variant is to decouple the resolution of the lighting pre-passes, from the final colour passes. In that scenario, the final color passes would need to 'scan' the lower resolution lighting buffers, for appropriate samples.

btw: god rays implementation

- completely 2d screenspace effect
- each light volume is rendered to an off-screen surface, one channel per light (in batches of 4)
  - pixel shader integrates light scattering along eye ray to first solid object (z buffer distance)
  - 3 pass 'smear' shader then smears dark regions with MIN blending
    - each pass doubles distance of smear

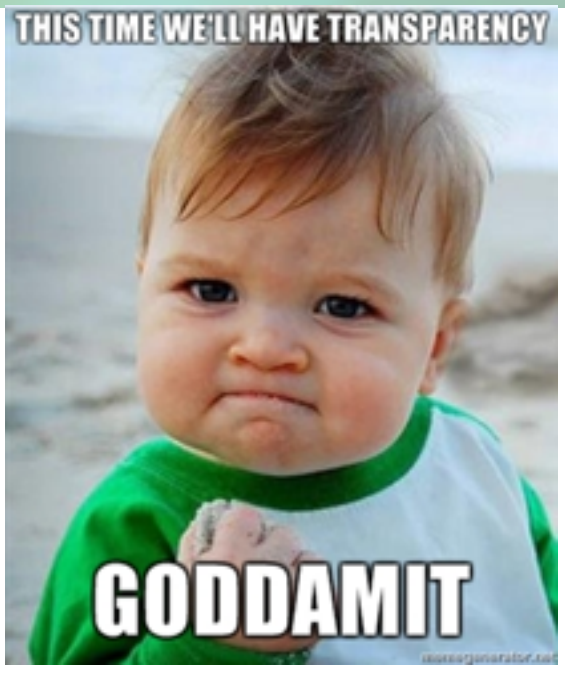


The smear shader is a simple 5 tap linear 2d 'smear': the screenspace pixel position is scaled towards the source of the light, and the samples merged together.

Rather than summing the samples, which leads to a soft edge on both sides of the smeared image, the sampled values are biased by the smear distance, and then min-blended.

Attempt 2: aka LBP2 ‘back to forward’

- wanted transparent surfaces, and particles.
- ...but now, no MSAA (MLAA instead)



Wednesday, 10 August 11

LBP2: want alpha BADLY. and particles. and MLAA -> no msaa sample rate.back to irradiance voumes!

- but now in world space. world is flat-ish
  - quantize to stop crawling
  - clamp maximum size
  - still sucks for oblique views

- first, voxelize the scene. render to 360p x 4MRTs -> 16 slices in one pass.

we didn't want accurate binary voxelisation, we wanted coverage per voxel. so alpha-blend z-ranges

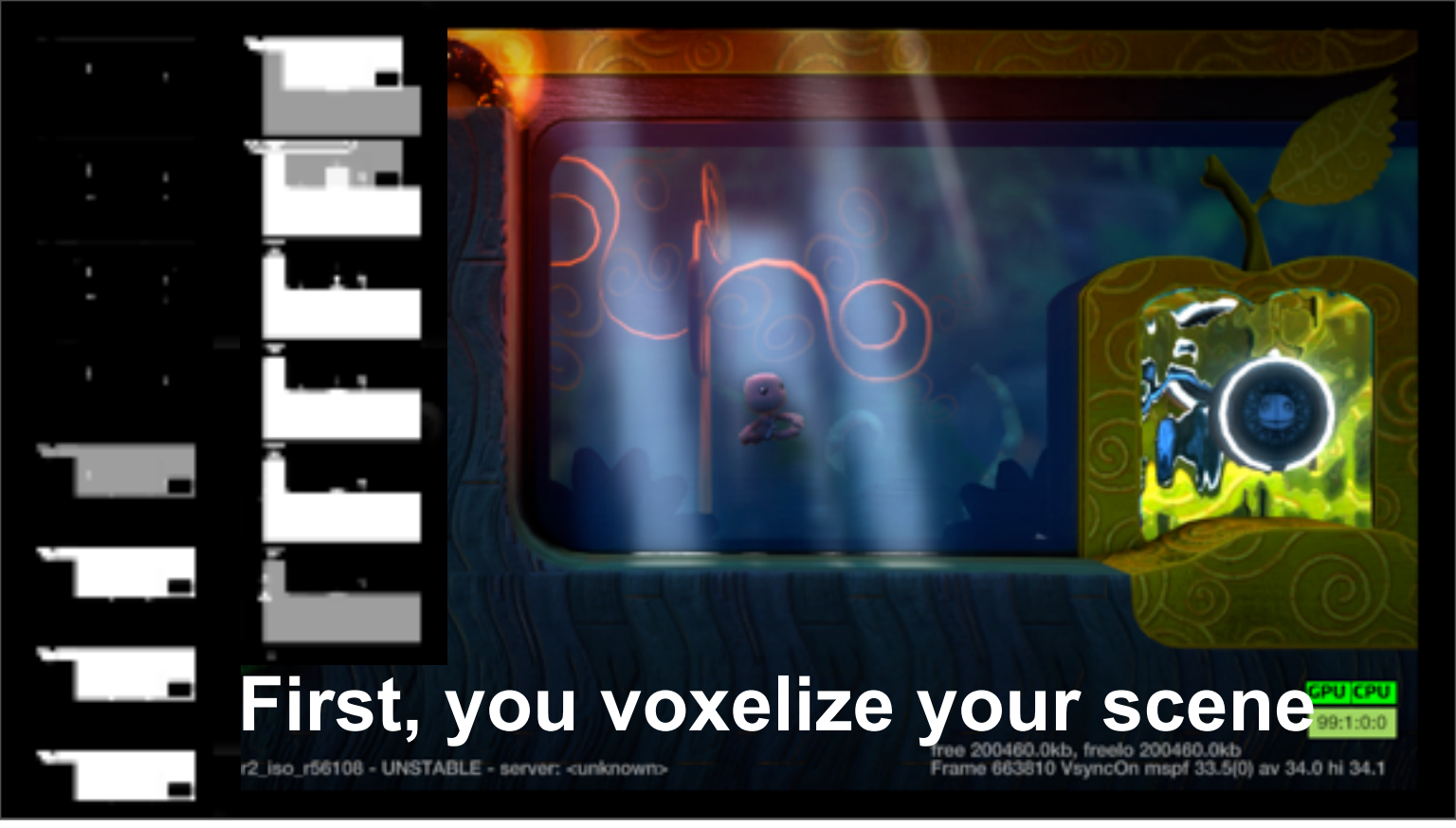
pixel shader computes front (from z) and back (by reflection or constant-per object) and outputs to MRT

this is an example of good enough: could have:

rendered back faces & subtracted

precomputed backfaces (limited rotation?)

raycast?!



Wednesday, 10 August 11

- first, voxelize the scene. render to 360p x 4MRTs -> 16 slices in one pass.

we didn't want accurate binary voxelisation, we wanted coverage per voxel. so alpha-blend z-ranges

pixel shader computes front (from z) and back (by reflection or constant-per object) and outputs to MRT

this is an example of good enough: could have:

rendered back faces & subtracted

precomputed backfaces (limited rotation?)

raycast?!

Wednesday, 10 August 11

- first, voxelize the scene. render to 360p x 4MRTs -> 16 slices in one pass.

we didn't want accurate binary voxelisation, we wanted coverage per voxel. so alpha-blend z-ranges

pixel shader computes front (from z) and back (by reflection or constant-per object) and outputs to MRT

this is an example of good enough: could have:

rendered back faces & subtracted

precomputed backfaces (limited rotation?)

raycast?!

- first, voxelize the scene. render to 360p x 4MRTs -> 16 slices in one pass.

we didn't want accurate binary voxelisation, we wanted coverage per voxel. so alpha-blend z-ranges

pixel shader computes front (from z) and back (by reflection or constant-per object) and outputs to MRT

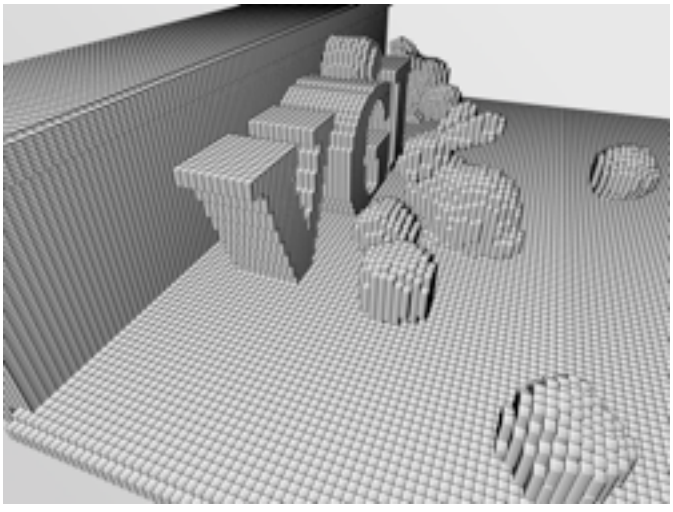
this is an example of good enough: could have:

rendered back faces & subtracted

precomputed backfaces (limited rotation?)

raycast?!

How we Voxelized the Scene



(Image from S. THIEDEMANN, N. HENRICH, T. GROSCH, S. MÜLLER 2011)

We could bit-slice it...
but we're more intersted in ***coverage***
so pixel shader should compute and α-blend 'z coverage'

We just used z buffer value and fixed thickness!
'good enough'

could have precomputed, or raycast, or mirrored, or...
...but fixed was fine for us. MOVING ON!

how we voxelized the scene

first we write to 4 MRTs, 16 slices packed:

```
  color_out0 = saturate(zthick-abs(float4(0,1,2,3)-zcen)) * alpha;
color_out1 = saturate(zthick-abs(float4(4,5,6,7)-zcen)) * alpha;
color_out2 = saturate(zthick-abs(float4(8,9,10,11)-zcen)) * alpha;
color_out3 = saturate(zthick-abs(float4(12,13,14,15)-zcen)) * alpha;
```

Then unswizzle, blur and downsample to a MIP pyramid
360p x 4 RGBA MRTS
360p x 16 greyscale slices
180p x 16 slices
90p x 8 slices
45p x 4 slices

Wednesday, 10 August 11

next we unpack/unswizzle the rgba channels into a volume tex compute an image pyramid aka mipchain

(first one doesnt reduce z so you get 180p x 16, 90p x 8, 45p x 4)

Wednesday, 10 August 11

now, splat lights into irradiance volume in same space (fp16, 180p - really blurry so thats ok)

- upgraded from just colour: stored somthing like 1st order SH in monochrome, with 0th order in RGB

- custom falloff textures - something like 1-d

- SH0 multiplied by N.SH1 - gives sharper falloff (1-d)^2

- was planning to then 'smear' this (propegate light) in multiple steps. ie scatter

however.. while prototyping, we discovered that you could just sample the voxel rep 16 times and it still ran plenty fast enough. YAY! SHADOWS!

Wednesday, 10 August 11

FOWARD rendering

- now forward rendering is trivial: just sample the irradiance volumes and do SH0.rgb * dot(N, SH1)
- you can even do specular!
- it's fast enough to do for particles too
- some examples
- you now have a fast samplable representation of the flow of light

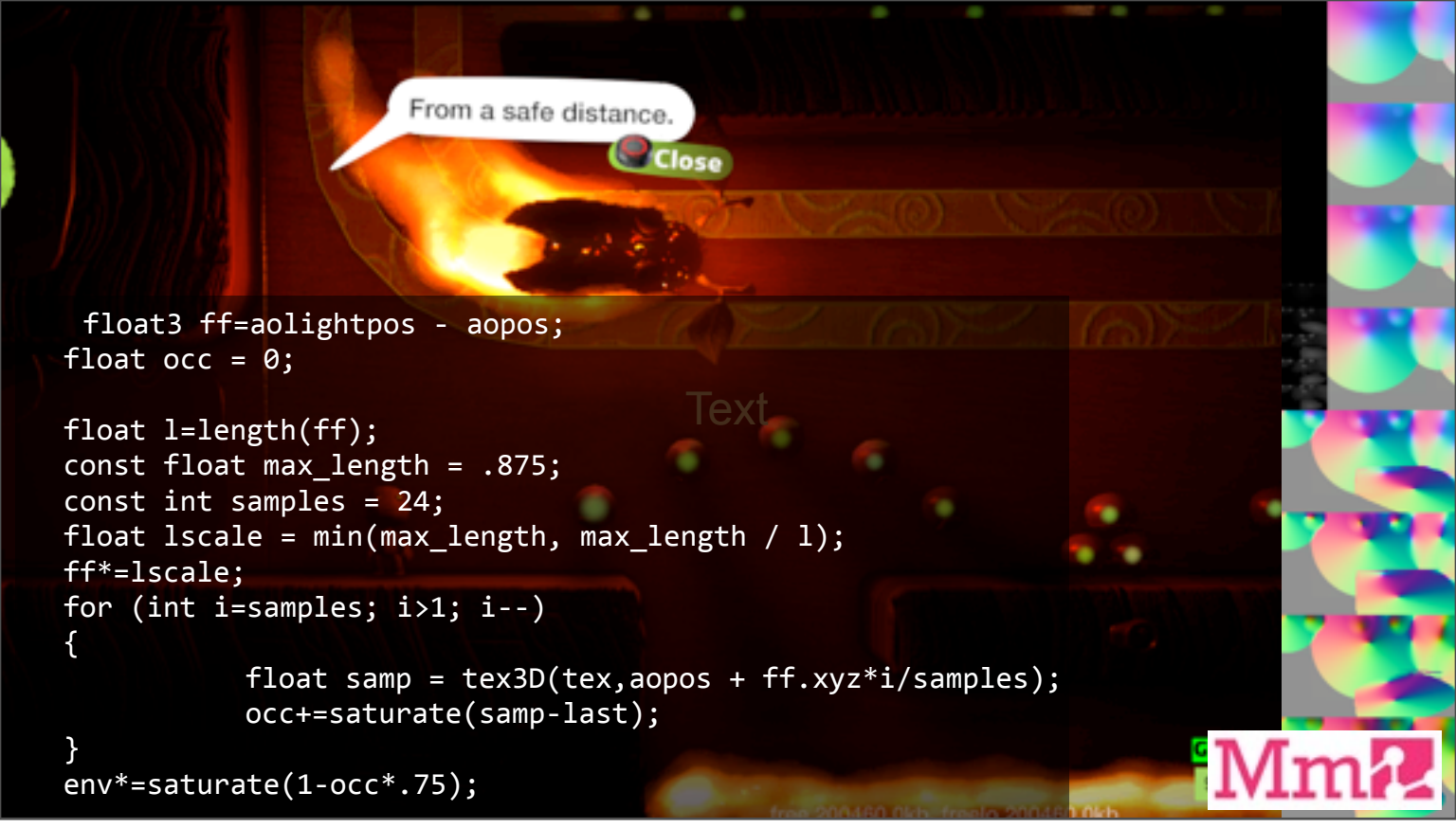
custom falloff baked in at ‘splat time’
something like (1-d)
at (forward) shade time, we do `SH0.rgb * dot(SH1.xyzw, N.xyzw)`
so you get falloff of (1-d)^2

- custom falloff textures - something like 1-d
- SH0 multiplied by N.SH1 - gives sharper falloff (1-d)^2

We were planning to ‘smear’/propagate the light.
However, in a quick hack we just sampled the voxel texture 24 times towards the light. It was fast!
Real raycast shadows FTW!

Wednesday, 10 August 11

- was planning to then 'smear' this (propagate light) in multiple steps. ie scatter however.. while prototyping, we discovered that you could just sample the voxel rep 16 times and it still ran plenty fast enough. YAY! SHADOWS!


From a safe distance.
Close
float3 ff=aolightpos - aopos;
float occ = 0;
Text
float l=length(ff);
const float max_length = .875;
const int samples = 24;
float lscale = min(max_length, max_length / l);
ff*=lscale;
for (int i=samples; i>1; i--)
{
float samp = tex3D(tex,aopos + ff.xyz*i/samples);
occ+=saturate(samp-last);
}
env*=saturate(1-occ*.75);
Mm?

Wednesday, 10 August 11

AO

you can also sample AO in world space – in your forward renderer, you sample the pyramid at increasing distance from the surface. similar to cone tracing in [gigavoxels]


WORLD SPACE
AO & sky light
AO

video of voxels for skylight

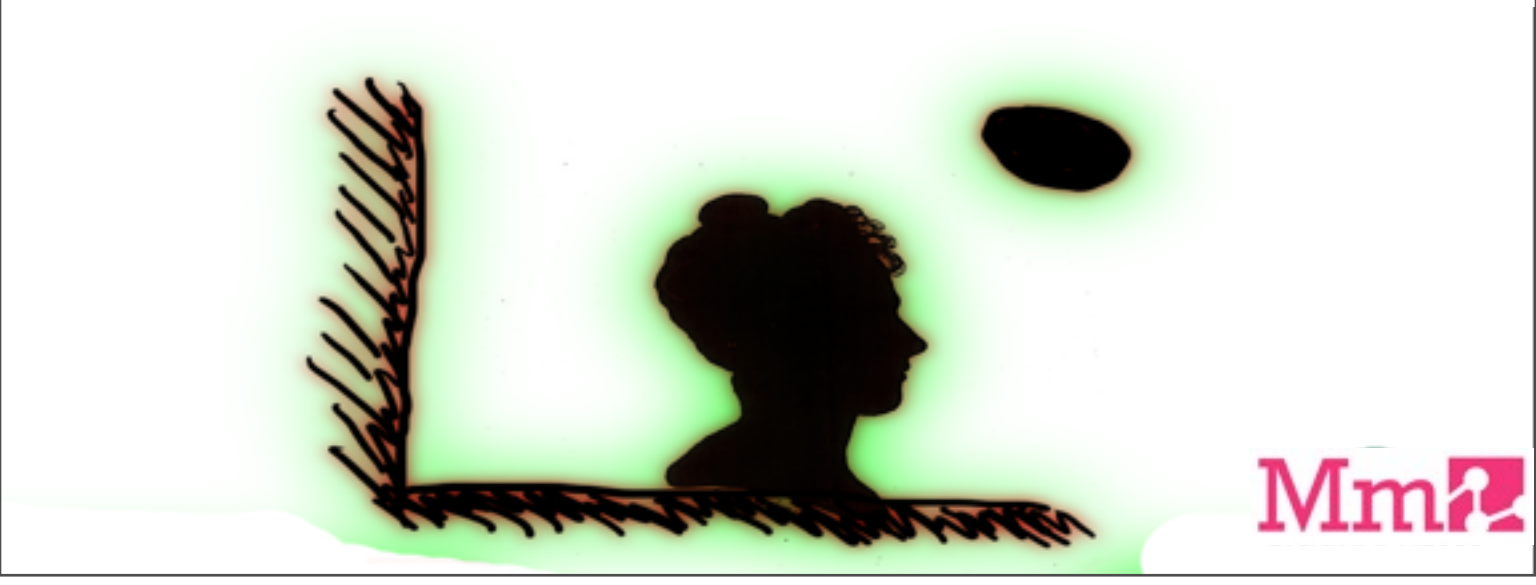
Sketch of what's going on

sorry for stealing from very old siggraph 2006 slides here. but it represents the idea.

given that you have an image pyramid of your world's voxelization 'coverage', you can make use of these mips by repeatedly sampling at a small distance (say, 1 voxel) away from the surface, and summing at all levels of the heirachy. instant AO!

Sketch of what's going on

Sketch of what's going on



Sketch of what's going on

Sketch of what's going on

Sketch of what's going on

Sketch of what's going on

...and by bending the normal towards the light, you get a skylighting effect, as objects 'under' others tend to feel the effects.

Sketch of what's going on

Sketch of what's going on

Code for world space AO/skylight


```
float DynamicAO(float3 vec2eye, half4 iNormal)
{...
       half4 aoNormal = iNormal;
       aoNormal.y+=0.75; // push towards the sky
       worldPos.xyz+=aoNormal.xyz;
        aoNormal.w=1;
       float ao=tex3Dlod(tex,worldPos).z;
       iNormal.xyz*=2;
       worldPos.xyz+=aoNormal.xyz;
       ao+=tex3Dlod(tex_lo,worldPos).z;
       iNormal.xyz*=2;
       worldPos.xyzw+=aoNormal.xyzw;
       ao+=tex3Dlod(tex_lo,worldPos).z;
       iNormal.xyz*=2;
       worldPos.xyzw+=aoNormal.xyzw;
       ao+=tex3Dlod(tex_lo,worldPos).z;
       return 1-ao*minaabb.w;
}
```

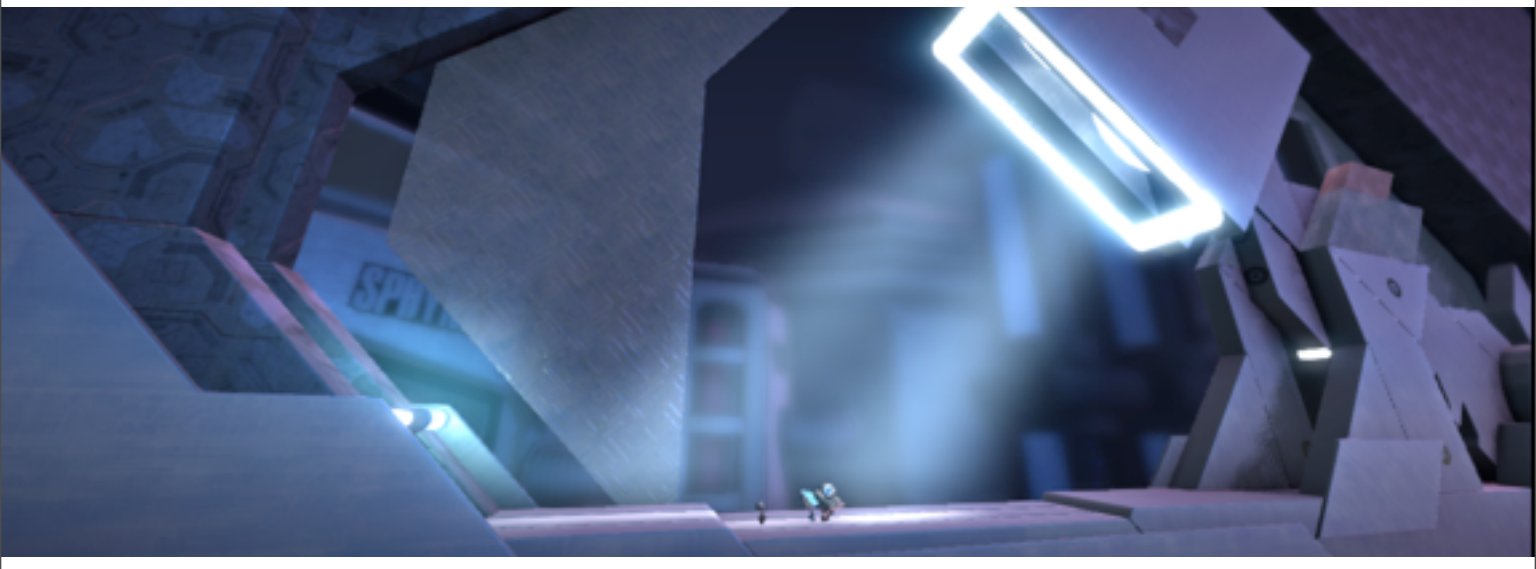
sneaky MIP-level picking in w

AO

you can also sample AO in world space – in your forward renderer, you sample the pyramid at increasing distance from the surface. similar to cone tracing in [gigavoxels]

Single scattering

probably the most defining thing of the LBP2 atmosphere/look



We just brute force raymarch through the SH0 light volume from eye to z buffer (at 160p, with 2x2 stratified jitter on the sample points)

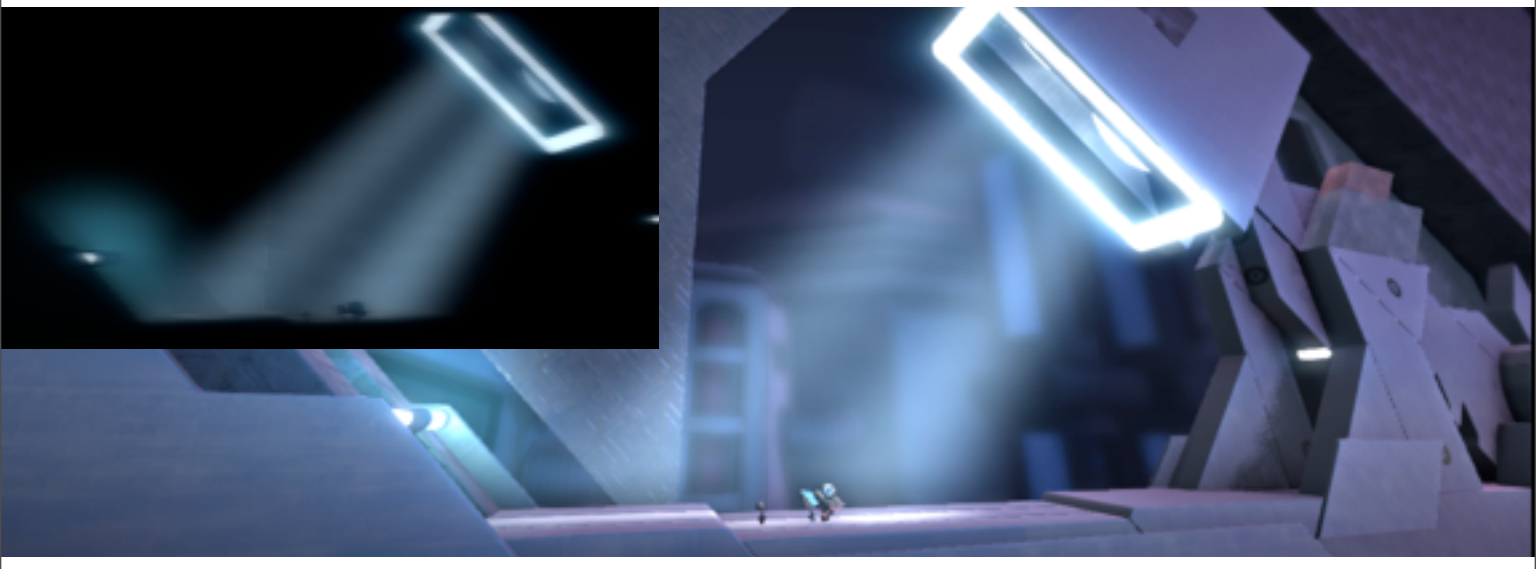
SINGLE SCATTERING

This actually is the most defining feature of lbp2's atmosphere.

at the end of the frame, we ray-march in screenspace rays from eye to z buffer, through SH0 volume.

do this at 160p and we jitter the rays on a 2x2 stratified grid, and then blur it out with a separable gaussian blur (also used for bloom). final comp controlled by user with 'foginess' slider.

Single scattering



then we comp it with our bloom buffer, and separably blur away the noise

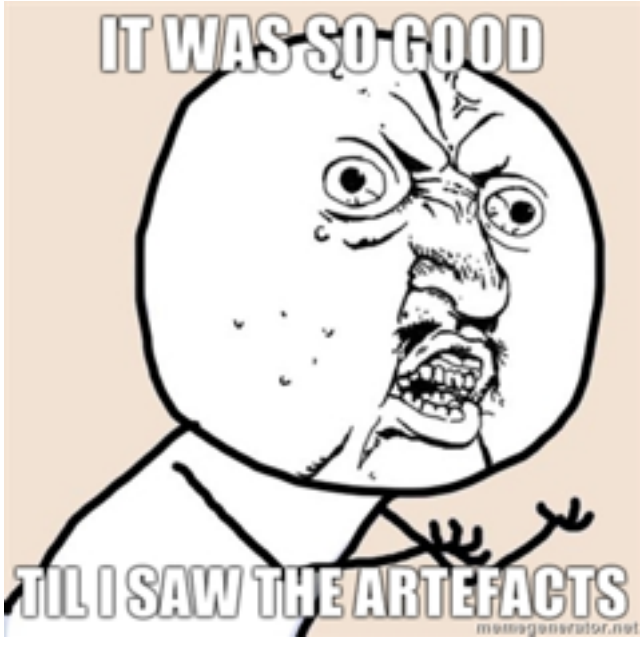
SINGLE SCATTERING

This actually is the most defining feature of lbp2's atmosphere.

at the end of the frame, we ray-march in screenspace rays from eye to z buffer, through SH0 volume.

do this at 160p and we jitter the rays on a 2x2 stratified grid, and then blur it out with a separable gaussian blur (also used for bloom). final comp controlled by user with 'foginess' slider.

This is awesome!



Wednesday, 10 August 11

THIS IS AWESOME!

but

downsides:

* super lo-res if the scene gets large

* not as fast per-light as it's gather (go back to scatter/smear?)

* low order SH-type effect gets confused with 'duelling lights' - especially for specular: red light on left, blue light on right, you get one pink specular hilight in the middle...

It has obvious limitations:

- super lo-res if the scene gets large - aliasing at oblique angles
- not as fast per-light as we'd like
  - beacuse of shadow raymarch gather
    - (could go back to scatter/smear?)

Wednesday, 10 August 11

THIS IS AWESOME!

but

downsides:

* super lo-res if the scene gets large

* not as fast per-light as it's gather (go back to scatter/smear?)

* low order SH-type effect gets confused with 'duelling lights' - especially for specular: red light on left, blue light on right, you get one pink specular hilight in the middle...

It has obvious limitations:

- low order SH-type effect gets confused with 'duelling lights'
  - especially for specular
  - red light on left, blue light on right, you get one pink specular hilight in the middle...

Wednesday, 10 August 11

THIS IS AWESOME!

but

downsides:

* super lo-res if the scene gets large

* not as fast per-light as it's gather (go back to scatter/smear?)

* low order SH-type effect gets confused with 'duelling lights' - especially for specular: red light on left, blue light on right, you get one pink specular hilight in the middle...

But also: obvious extensions!

SIGGRAPH2011

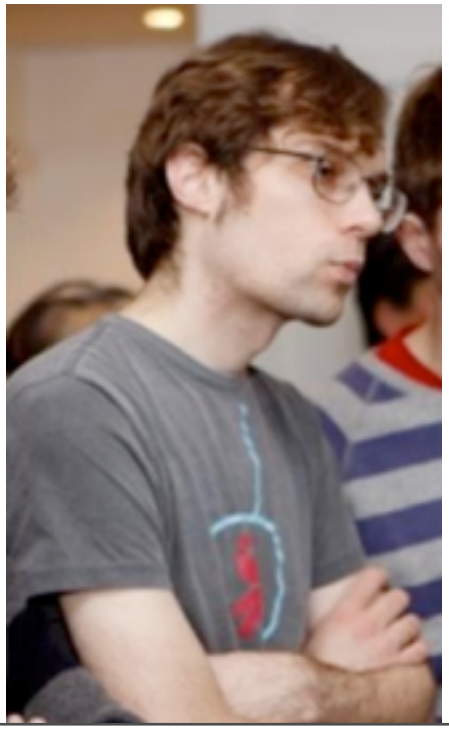
- Things we didn’t even bother trying due to time constraints:
- adaptive resolution - quadtree?
- or, cascade of resolutions centered on camera?
- light propagation instead of raymarching?

Wednesday, 10 August 11

extensions we didn't even try but are obvious:

* adaptive resolution – quadtree?
* or, cascade of resolutions centered at camera?
* light propegation instead of shadow gather

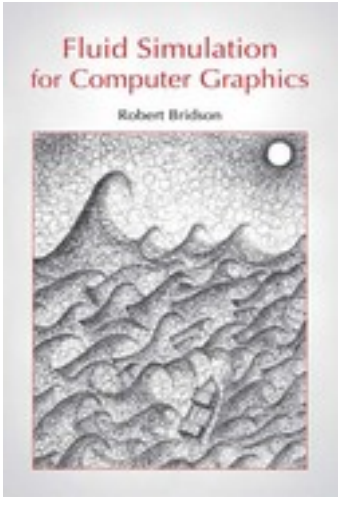
Particles & Fluids + Voxels = win



- The main graphics programmer for LBP2 was Anton Kirczenow.
  - I'll now pretend to be him :)

Fluids in LBP2

- Fluids in LBP are basically based on a FLIP integrator as described in R. Bridson's book.
  - Coded as 47 different SPU jobs!
  - Essentially, 100s of kernels run sparsely over the fluid particles and/or grids
  - Very SPU friendly - lightly vectorized in a few places, unrolled a few loops
    - but didn't have to downcode to ASM!
    - we end up GPU fillrate bound anyway



Aside on implementing SPU kernels

- a little bit of PERL and C++ preproc goes a long way!
- Each SPU job ‘tagged up’ in a .h file
  - included in multiple places:
    - PPU side, collect main memory pointers, sizes and access patterns, decide how many jobs to split into, put jobs into a job chain
    - SPU side, unpack DMA inputs into arguments to the kernel function
    - SPU side, generate the kernel code and call it
    - SPU side, make DMA outputs go

Aside on implementing SPU kernels


```
-  #ifdef WANT_CODE
void
#endif // WANT_CODE
#ifdef WANT_NAME // _FUNCTION_NAME
FlipPremultPoisson
(
#endif // _FUNCTION_ARGS
      STREAM_WRITE( float, poisson1, poisson1_size )
      STREAM_WRITE( float, poisson2, poisson2_size )
      STREAM_WRITE( float, poisson3, poisson3_size )
      STREAM_READ( float, preconditioner, preconditioner_size )
      STREAM_READ( v4, poisson, poisson_size )

      STREAM_END
)
```

Aside on implementing SPU kernels


```
	#ifdef WANT_CODE
)
{
	u32 np = stream_end_decl;
	for (u32 i=0; i<np; i++)
	{
		float precond = preconditioner[i];
		v4 pois = poisson[i];

		poisson1[i] = pois.getY()*precond;
		poisson2[i] = pois.getZ()*precond;
		poisson3[i] = pois.getW()*precond;
	}
}
#endif // WANT_CODE
```

a list of all the kernels!


```
flip_accumulate.h
flip_add_gradient.h
flip_add_gravity.h
flip_alive.h
flip_apply_boundary_markers.h
flip_apply_boundary_velocity.h
flip_apply_poisson.h
flip_apply_preconditioner1.h
flip_apply_preconditioner2.h
flip_average.h
flip_barify.h
flip_center.h
flip_check.h
flip_compress_markers.h
flip_copy.h
flip_dot.h
flip_find_divergence.h
flip_form_poisson.h
flip_form_preconditioner.h
flip_generate.h
flip_generate_vel.h
flip_get_velocity_update.h
flip_grad.h
flip_increment.h
flip_index.h
flip_keys.h
flip_kick.h
flip_mag.h
flip_move_particles_in_grid1.h
flip_move_particles_in_grid2.h
flip_order.h
flip_premult_poisson.h
flip_save_velocities.h
flip_scale.h
flip_scale_and_increment.h
flip_sort.h
flip_spawn.h
flip_splat_fluid_markers.h
flip_stagger.h
flip_sum_divergence.h
flip_trilerp_velocities.h
flip_unscale.h
flip_update_from_grid.h
flip_vert_swizzle.h
flip_vorticity.h
flip_vorticity_force.h
flip_zero.h
```

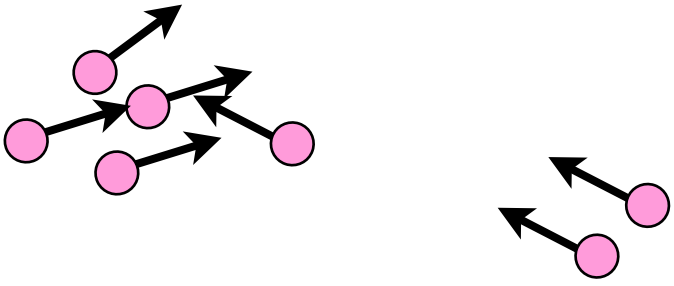
SPU jobs interleave with other work

An overview of FLIP

- FLIP blends the advantages of particle & grid based fluid solvers
  - so it's a perfect fit to use the voxelization described earlier to provide collision info
  - the voxels will also be used in lighting the particles
  - even with very low-resolution grids, you get hardly any dissipation -> nice looking fluids

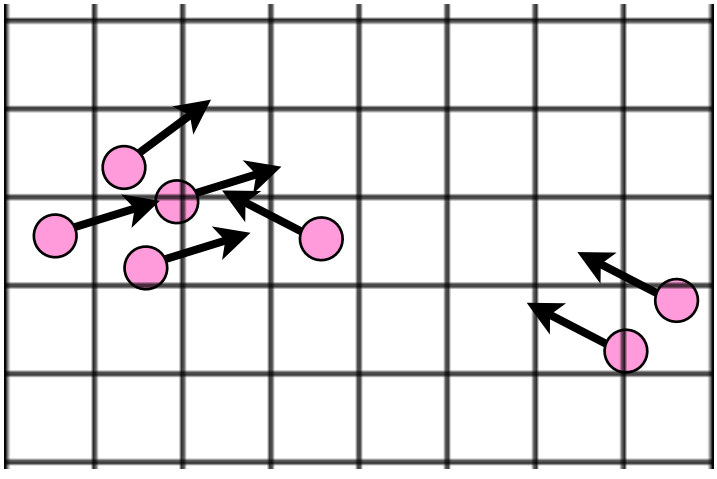
An overview of FLIP

- Start with some particles with velocities...



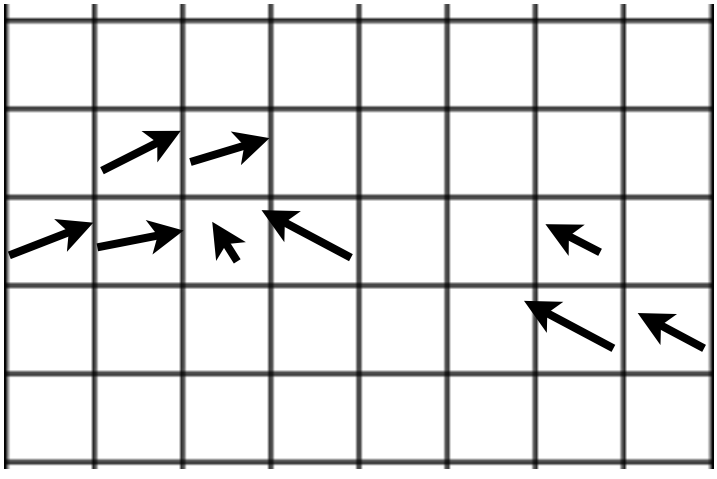
An overview of FLIP

- project their velocities onto a lo-res (3D) grid



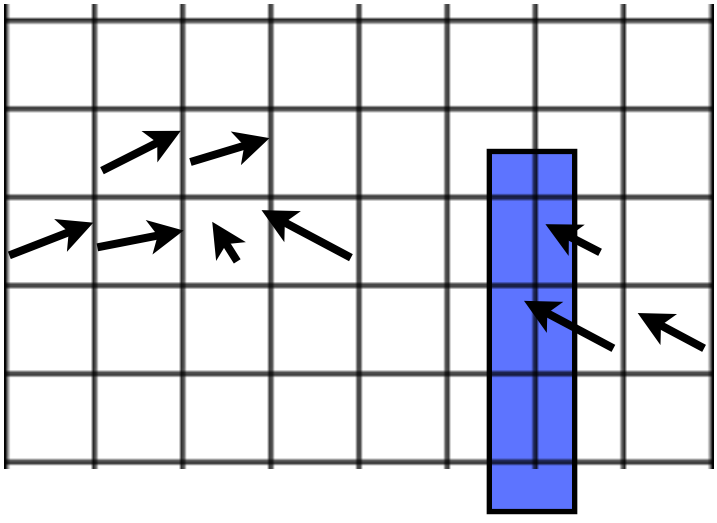
An overview of FLIP

- project their velocities onto a lo-res (3D) grid



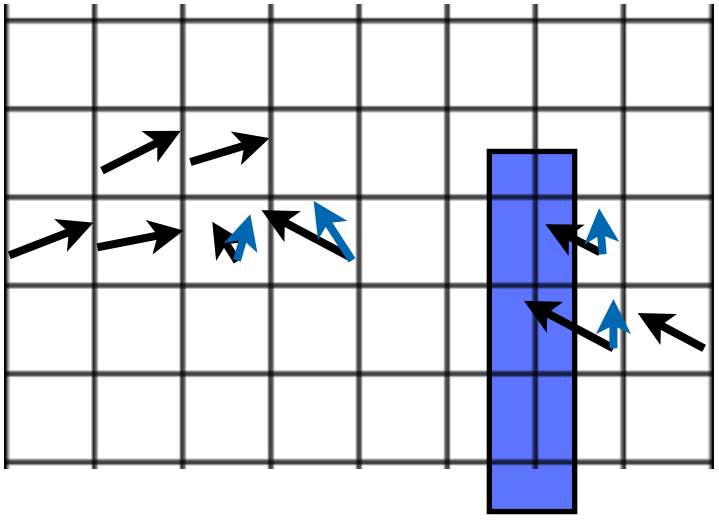
An overview of FLIP

- downsample world voxel grid for collision



An overview of FLIP

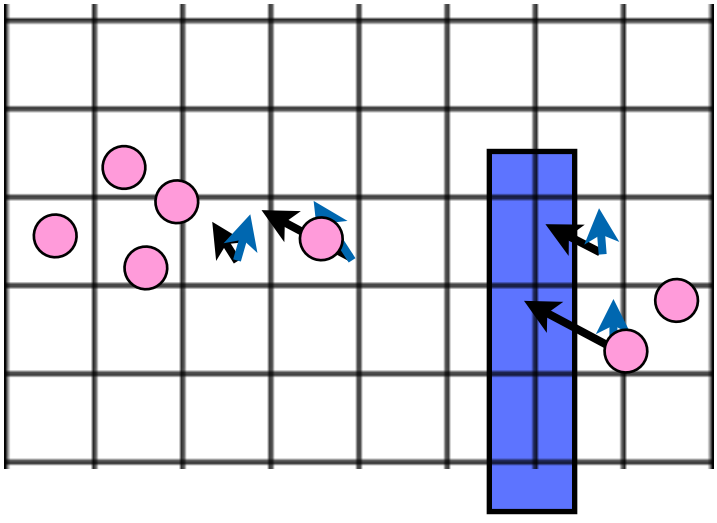
- fix up velocities with preconditioned conjugate gradient solver



- also handle collision
- throw in vorticity confinement
  - especially for fire

An overview of FLIP

- compute changes in velocity, and re-apply to particles



- its this coupling between grid & particles that keeps detail while allowing solve to be cheap
- we use 16384 particles, only 64x32x8 grid!

Implementation Details

- Particles moved with 2nd order Runga-Kutta
- RSX bound despite rendering at 1/2 res
- Particles & Volume track camera
  - easy for us as 2.5D
  - some dodgy hacks when the camera zooms
    - grid changes scale, fixed resolution
    - have to fake it to make gravity appear constant
- Run two independent simulations
  - ‘water type’, ‘gas type’

Gas cloud particles

- Voxel volumes used for lighting & soft z; FLIP used for motion; 1/2 res, upsampled in post
- SPU sorts particles using AASort

Spawn lots of fairly short lived particles randomly from polygon cloud shapes with uniform-ish density. Ramp the alpha to fade in/out over particle lifetime.
Fluid sim for swirly motion.
To render: SPU sorts particles front to back (with AASort).
Comp lots of alpha sprites over top of each other. Use RSX MSAA hack to do it at ¼ res because otherwise it absolutely kills the fill rate.
Sample the lighting volumes in the sprite shader to pick up glow, shadows etc.
Use the AO volume to multiply through the alpha channel to do the “soft Z” trick to get rid of those hideous sprite/mesh intersections. (alpha *= 1-solidness of voxel scene at sprite pos)
Upscale and comp it over the main scene, hope no one notices the haloing.

Gas cloud particles shadows

- gas renders into variance shadow maps
  - so as to not look ‘pasted on’
- 2 hacks to allow for transparency in VSM:
  - 50% stipple
  - shrink particles as they fade

Dissolve

Pretty much like gloop case but mesh particles turn into gas and fade away.
However we also render variance shadow maps for the particles to not look “pasted on”.
Problem: Variance does not do transparency for shadows.
Hack 1: Stipple pattern for particle transparency level! Kinda works, if you don’t look too closely.
50% stipple so particle shadows aren’t as strong as mesh shadows (looks weird otherwise).
Hack 2: Shrink the particles that cast shadows by their alpha level – shadows smoothly disappear with the particles.

Fire/Flamethrowers/Thrusters

Spawn lots of high velocity particles, pick up nice motion from fluid sim.

Use vertex shader to extrude particle along velocity to do motion blur/hide strobing.
Crossfade to a smokey particle over the lifetime.
Splat the emissive part of the particle into 4 channel texture – split on Z values, additively accumulate grayscale “hotness” for each of 4 layers.
Particle comp pass converts hotness to a blackbody color texture, adds it on top of smokey particles renderered with usual alpha blending.
Post FX uses the “hotness” for heat haze effect strength.
Rendering is working quite hard to try to hide the particle nature of the system. ☹

Fire/Flamethrowers/Thrusters

Spawn lots of high velocity particles, pick up nice motion from fluid sim.

Use vertex shader to extrude particle along velocity to do motion blur/hide strobing.
Crossfade to a smokey particle over the lifetime.
Splat the emissive part of the particle into 4 channel texture – split on Z values, additively accumulate grayscale “hotness” for each of 4 layers.
Particle comp pass converts hotness to a blackbody color texture, adds it on top of smokey particles renderered with usual alpha blending.
Post FX uses the “hotness” for heat haze effect strength.
Rendering is working quite hard to try to hide the particle nature of the system. ☹

Fire/Flamethrowers/Thrusters

Spawn lots of high velocity particles, pick up nice motion from fluid sim.

Use vertex shader to extrude particle along velocity to do motion blur/hide strobing.
Crossfade to a smokey particle over the lifetime.
Splat the emissive part of the particle into 4 channel texture – split on Z values, additively accumulate grayscale “hotness” for each of 4 layers.
Particle comp pass converts hotness to a blackbody color texture, adds it on top of smokey particles renderered with usual alpha blending.
Post FX uses the “hotness” for heat haze effect strength.
Rendering is working quite hard to try to hide the particle nature of the system. ☹

Fire/Flamethrowers/Thrusters

Spawn lots of high velocity particles, pick up nice motion from fluid sim.

Use vertex shader to extrude particle along velocity to do motion blur/hide strobing.
Crossfade to a smokey particle over the lifetime.
Splat the emissive part of the particle into 4 channel texture – split on Z values, additively accumulate grayscale “hotness” for each of 4 layers.
Particle comp pass converts hotness to a blackbody color texture, adds it on top of smokey particles renderered with usual alpha blending.
Post FX uses the “hotness” for heat haze effect strength.
Rendering is working quite hard to try to hide the particle nature of the system. ☹

Fire/Flamethrowers/Thrusters

Spawn lots of high velocity particles, pick up nice motion from fluid sim.

Use vertex shader to extrude particle along velocity to do motion blur/hide strobing.
Crossfade to a smokey particle over the lifetime.
Splat the emissive part of the particle into 4 channel texture – split on Z values, additively accumulate grayscale “hotness” for each of 4 layers.
Particle comp pass converts hotness to a blackbody color texture, adds it on top of smokey particles renderered with usual alpha blending.
Post FX uses the “hotness” for heat haze effect strength.
Rendering is working quite hard to try to hide the particle nature of the system. ☹

Fire/Flamethrowers/Thrusters

- Emissive particles split into 4 layers
  - splat heat into 4 channel texture
  - post shader composites using blackbody palette
  - post also uses total heat for ‘heat-haze’ effect
  - particles cross fade to smoke / debris
  - hot objects splat their shape into the light volume textures to create local ‘glow’ lights
  - was quite difficult to hide the ‘particle’ nature of the system

Spawn lots of high velocity particles, pick up nice motion from fluid sim.

Use vertex shader to extrude particle along velocity to do motion blur/hide strobing.
Crossfade to a smokey particle over the lifetime.
Splat the emissive part of the particle into 4 channel texture – split on Z values, additively accumulate grayscale “hotness” for each of 4 layers.
Particle comp pass converts hotness to a blackbody color texture, adds it on top of smokey particles renderered with usual alpha blending.
Post FX uses the “hotness” for heat haze effect strength.
Rendering is working quite hard to try to hide the particle nature of the system. ☹

Dissolve / Gloop / Water

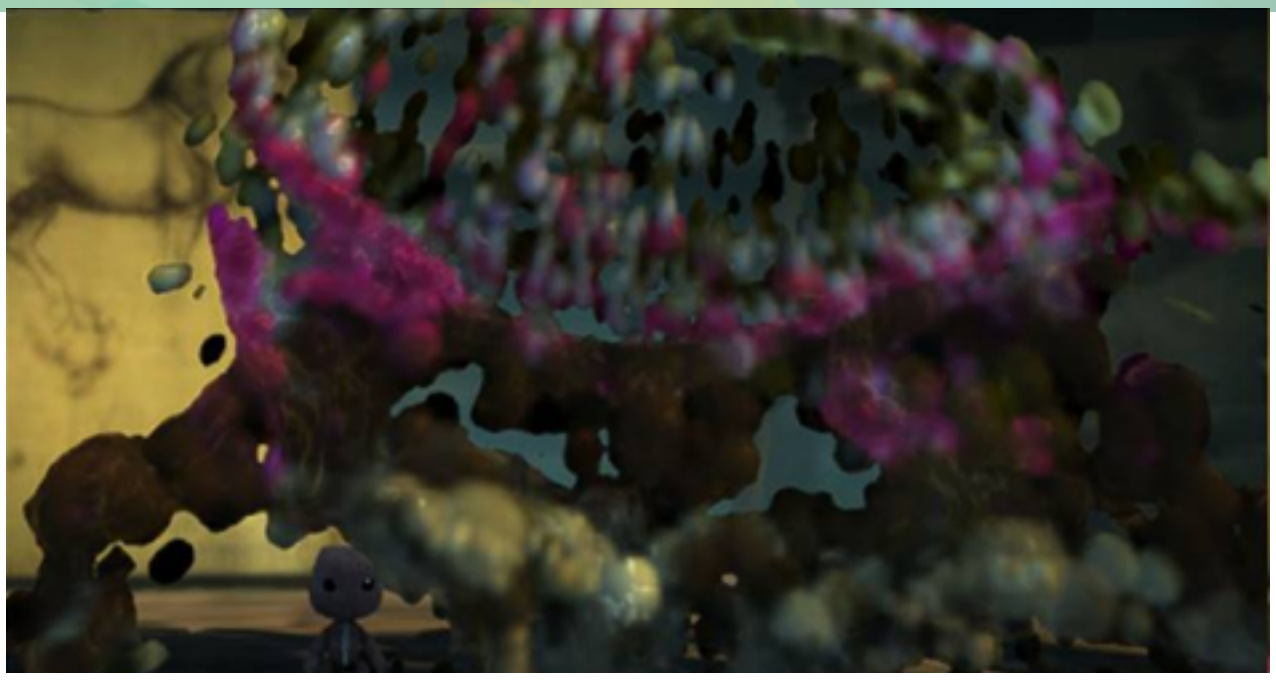
Dissolve / Gloop / Water

Dissolve / Gloop / Water

Dissolve / Gloop / Water

Gloop / Water

Gloop / Water

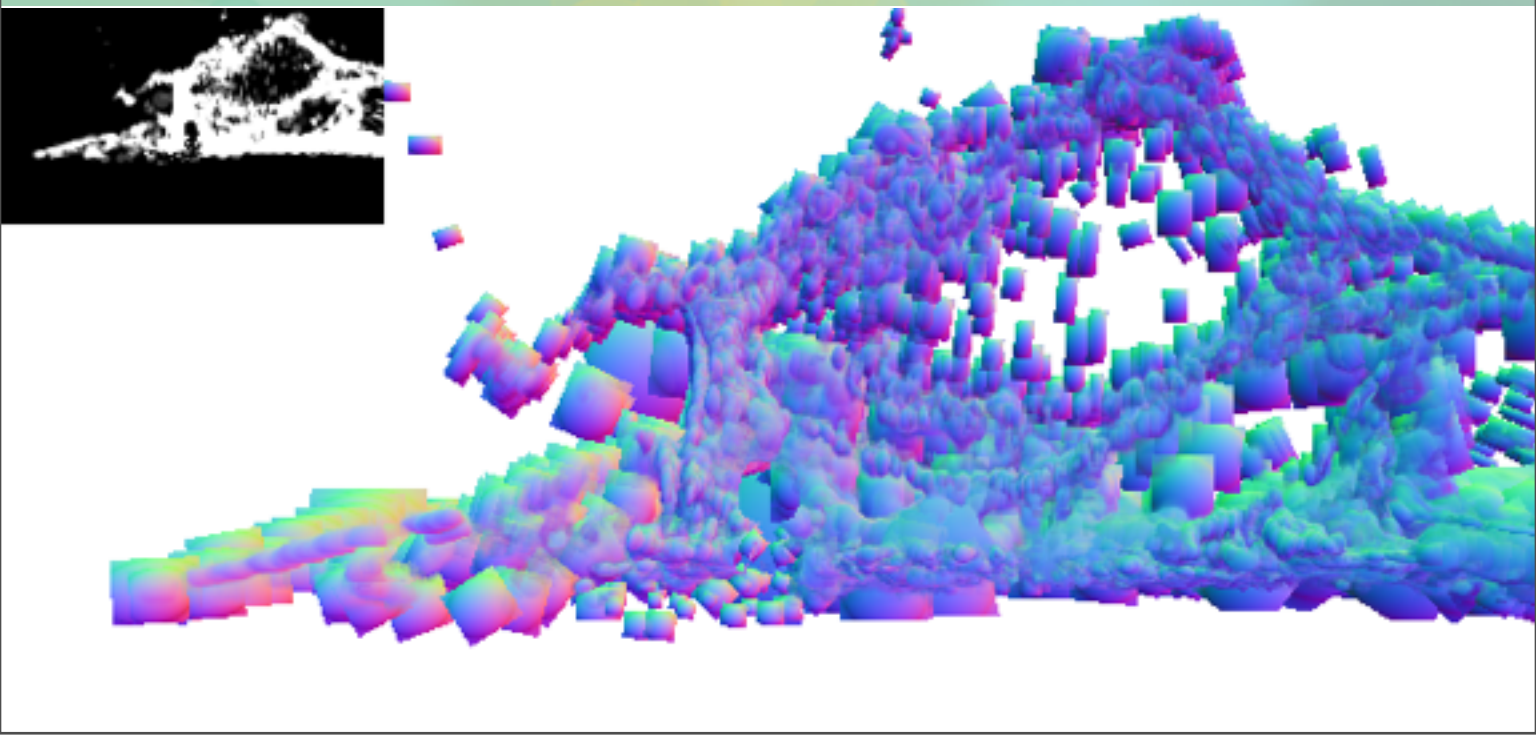


Gloop / Water

Gloop / Water

- Spawn particles
  - randomly distributed over mesh surface
  - render pointsprites with ‘normal’ shader
  - give initial velocity from rigid body + ‘explosiveness’
- Gloopy particles rendering was tricky
  - render normals & unlit color to 1/2 res RTs
  - careful thresholding of alpha channel to give art-directed balance of soft/hard edge & rim light

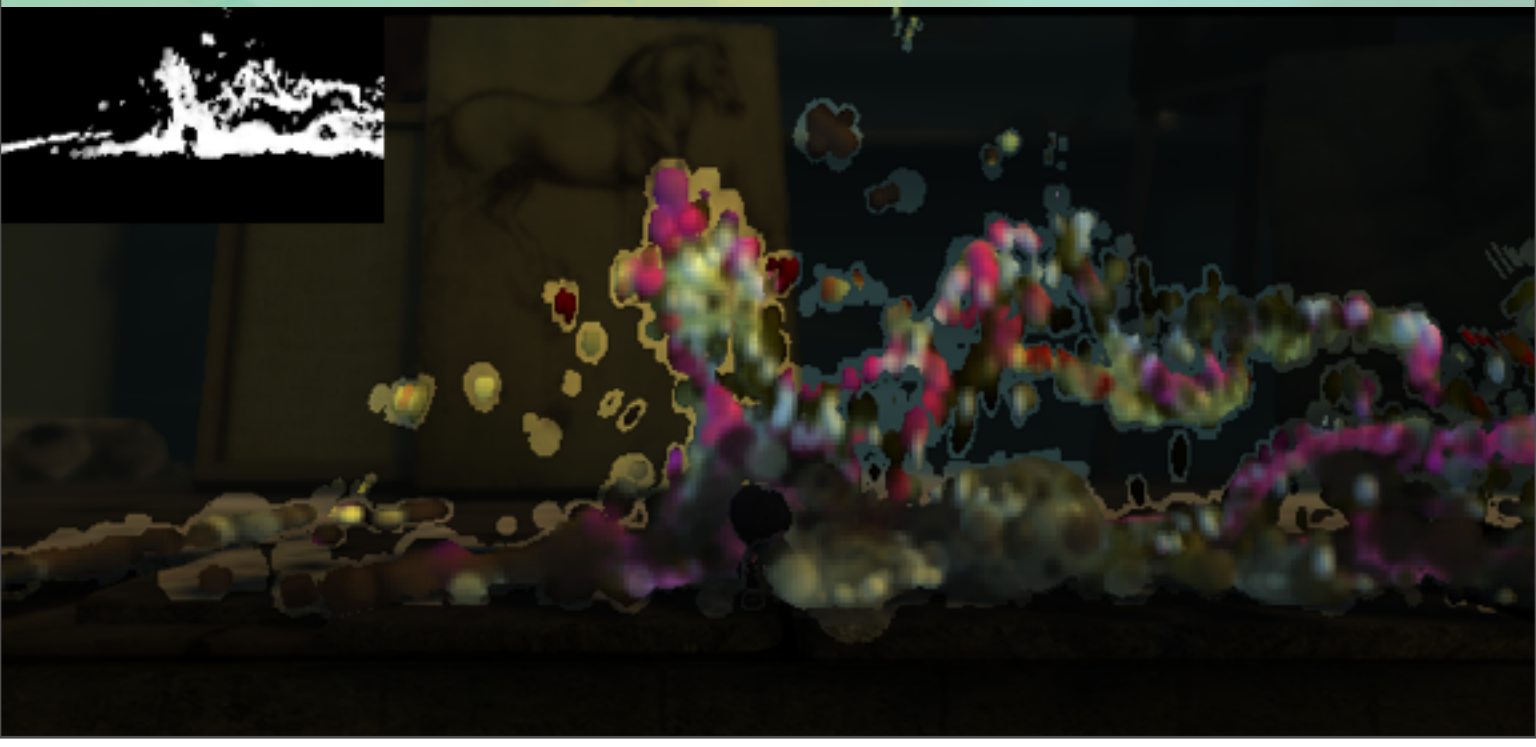
Gloop / Water



Run a 2nd version of the fluid sim in “water” mode in parallel with the “gas” mode sim!
Can turn an object into goop or gas – randomly sprinkle particles over the surface of a mesh, give an initial velocity along the normal. Tune for desired explosiveness.
Drop particles into the fluid sim.

Rendering is a major pain in the @$$!
Render point sprites sampling all the fancy materials/lighting/textures/shaders/stickers at the mesh surface points collecting the results to another rendertarget to use for particle color.
(Without writing yet another permutation of those materials!!)

Gloop / Water



This is as old as the hills!
I ♥ VOXELS and particles!

SIGGRAPH2011

- Don't underestimate how nice it is to have a hardware-samplable, filterable, heirachical scene coverage representation.
  - This sort of thing has been used for ages for things like god rays, lighting volume renders, lighting participating media
    (eg CUDA SDK sample)...

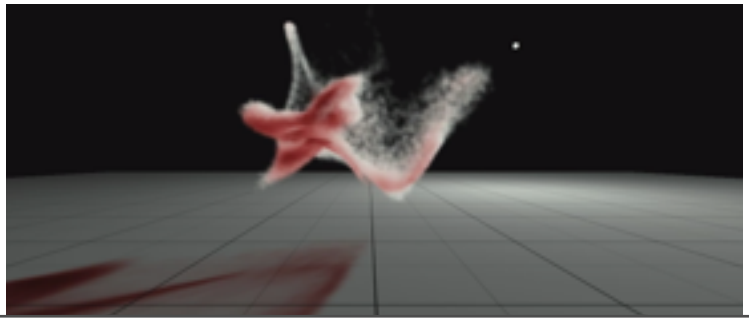


image from http://www.kinect-hacks.com/kinect-hacks/2011/03/12/cuda-sdk-smokeparticles-example-using-kinect

Mm?

Wednesday, 10 August 11

final aside: you can't underestimate how nice it is to have a hardware-samplable (efficient), filterable, heirachical representation of scene geometry. lots of awesome stuff in this space, see eg gigavoxels, i3d global illumination paper, but... this is as old as the hills!

can't find the reference, but this algorithm has been used to light participating media, and sky lights:

start by once again voxelizing your scene.

work through the scene in slices, filling in your 'skylight irradiance volume' one slice at a time.

start at the top (sky) most slice; set it all white. blend in that slice of your scene's voxelisation, to make occluders in that slice go black. (can also have emitters here...)

now, move down to the next slice by blurring and fading the last slice very slightly. repeat.

render scene using resulting skylight volume. YUM. (extra credit: sample volume a few times, off the surface, as per the AO in lbp2).

extra extra credit: have several passes through the volume in different directions, propegating light between layers as you go. eg could alternate over the cardinal directions, or up/down, or...

note that this approach can be amortised over multiple frames. stuff that propegates within a frame, 'travels at the speed of light' (in my terminology); stuff that takes several frames travels slower ('speed of sound'). Not everything has to go at the speed of light to look good.

food for thought, eh..!

Thanks!

SIGGRAPH2011

- I hope that gave you some food for thought.
  - simple techniques can give you a lot!
- References are in the slide notes (will be online).
- Thanks to Anton Kirczenow for most of the LBP2 gfx code.
- Thanks to Natalya Tatarchuk for inviting me to the course
- Thanks to MM and Sony WWS for all their hard work on LBP!
- Follow me on twitter! @mmalex @mediamolecule

Mm

references:

S. THIEDEMANN, N. HENRICH, T. GROSCH, S. MÜLLER 2011 Voxel-based Global Illumination
ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (i3D 2011), San Francisco, USA, 2011 (to appear)

BRIDSON, R. 2009. Fluid Simulation For Computer Graphics.
A.K Peters.

CRASSIN, C. , NEYRET, F. , LEFEBVRE, S., AND ISEMANN, E . 2009 Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering.
I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, ACM.

DACHSBACHER, C. ,AND STAMMINGER, M. 2006. Splatting indirect illumination.
I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM

EISEMANN, E. , AND DÉCORET, X. 2006 Fast scene voxelization and applications. In ACM SIGGRAPH Symposium on Interactive
3D Graphics and Games, ACM, 71-78.

EVANS, A., 2006 Fast approximations for global illumination on dynamic scenes
ACM SIGGRAPH 2006 Courses - Advances in Real-Time Rendering in 3D Graphics and Games Course

EVANS, A., 2009 LittleBigPlanet - Rendering Post-Mortem
ACM SIGGRAPH 2009 Courses - Advances in Real-Time Rendering in 3D Graphics and Games Course

References

S. THIEDEMANN, N. HENRICH, T. GROSCH, S. MÜLLER 2011 Voxel-based Global Illumination

ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (i3D 2011), San Francisco, USA, 2011 (to appear)

BRIDSON, R. 2009. Fluid Simulation For Computer Graphics.

A.K Peters.

CRASSIN, C. , NEYRET, F. , LEFEBVRE, S., AND ISEMANN, E . 2009 Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, ACM.

DACHSBACHER, C. ,AND STAMMINGER, M. 2006. Splatting indirect illumination.

I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM

EISEMANN, E. , AND DÉCORET, X. 2006 Fast scene voxelization and applications. In ACM SIGGRAPH Symposium on Interactive

3D Graphics and Games, ACM, 71-78.

EVANS, A., 2006 Fast approximations for global illumination on dynamic scenes

ACM SIGGRAPH 2006 Courses - Advances in Real-Time Rendering in 3D Graphics and Games Course

EVANS, A., 2009 LittleBigPlanet - Rendering Post-Mortem

ACM SIGGRAPH 2009 Courses - Advances in Real-Time Rendering in 3D Graphics and Games Course

GREGER, G. , SHIRLEY, P. , HUBBARD, P. M. , AND GREENBERG, D. P. 1998 The irradiance volume.

IEEE Computer Graphics and Applications 18, 32-43.

KAPLANYAN, A., 2009. Light propagation volumes in cryengine 3.

ACM SIGGRAPH 2009 Courses - Advances in Real-Time Rendering in 3D Graphics and Games Course

KAPLANYAN, A. , AND DACHSBACHER, C. 2010 Cascaded light propagation volumes for real-time indirect illumination.

I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ACM

RITSCHEL, T. , GROSCH, T. , KIM, M . H. , SEIDEL, H.-P. ,DACHSBACHER, C. , AND KAUTZ, J. 2008 Imperfect shadow maps for efficient computation of indirect illumination.

ACM Transactions on Graphics, in Proceedings of SIGGRAPH ASIA 2008