Rendering in Marmoset Toolbag

A Game Engine For Artists

Penn Engineering CIS 700
April 28, 2014

Marmoset

- Hi, I'm Jeff Russell (@j3ffdr)
- I work at Marmoset (www.marmoset.co)
  - Started early 2012
  - 3 partners + 1-3 contractors
  - All remote
  - The ‘t’ is not silent
  - We do graphics stuff!

Marmoset Toolbag

- Toolbag is a standalone real-time renderer for artist use
- “Game Engine for Artists”
- WYSIWYG final renders
- Basic scene, lighting, material editing
- Used for sharing, preview, presentation, material setup
- Output options for high quality screenshots, turntables
  - More to come ;)

Marmoset Toolbag

- Grew from a simpler in-house game dev tool
  - “Darkest of Days” at 8monkey
- Joe Wilson: “You guys should sell this”
- Initially free, now $129 / seat
- We’ve grown mainly by word of mouth
  - www.polycount.com

Marmoset Toolbag

- Version 2 (December 2013) is a full rewrite
- Windows 7/8 and OS X 10.9
- 64-bit only, C++
- Rough size:
  - Toolbag: ~60k lines
  - libCPR: ~15k
  - Shaders: ~5k
  - Server: ~3k

Demo!

Render Overview

- Toolbag 2 is a forward renderer (mostly), not deferred
- Need the flexibility of completely different shaders
  - skin, cloth, anisotropy, different spec models
- Dynamic light sources and shadow casting is "sort of deferred"

Render Order

1. Depth/normal pass
2. Reflection ray casts
3. SSAO
4. Main IBL pass
5. Reflection ray gather & composite
6. Dynamic lights & shadows
7. Wireframes & misc.
8. Post effects, AA & present

Depending on settings, only 4 & 8 will always run

Material System

- Swappable subroutines for various shading inputs
- Subroutines mainly focus on varying workflows
  - e.g. spec mask vs. metalness vs. IOR
- UI maps to shader subroutines more or less directly
- NOT a full shader graph system
  - User-friendly and approachable is a primary design goal
- Several million shader permutations
  - Requires runtime compilation as user edits
  - HLSL builds really slowly, especially loops >:|
  - If I were Microsoft I would work on this, but hey
  - Hash & cache to disk for later reuse, so only first hit is slow

Material System

- Quick list of some current modules:
  - Subdivision: Flat, PN Triangles
  - Displacement: Height, Vector
  - Surface: Normal Map, Detail Normal Map, Traced Parallax
  - Microsurface: Gloss/Roughness
  - Albedo: Albedo Map, Vertex Color
  - Diffusion: Unlit, Lambertian, Microfiber, Skin
  - Reflectivity: Specular Map, Matlaness Map, IOR
  - Reflection: Mirror, Blinn-Phong, Anisotropic
  - Occlusion: Occlusion Map, Vertex AO
  - Emissive: Emissive Map, Heat
  - Transparency: Cutout, Dither, Blend
  - + some extras (e.g. Dota 2, Substance support)

CPR

- Toolbag uses Direct3D 11 (Win) & OpenGL 3/4 (Mac)
- Future APIs?
  - GLES, WebGL, Consoles? Direct3D 12? OpenGL 5??
- Need to avoid rewriting everything for each platform!
  - Including shaders!
- How can we do this?

CPR

- CPR (Cross-Platform Render)
- In-house graphics abstraction library
- Supports backends for:
  - Direct3D 11
  - OpenGL 3
  - OpenGL ES 2
  - No-Op
- Recently dropped (we consider these dead):
  - Direct3D 9 (win + xbox 360)
  - OpenGL 2
  - GCM (ps3)

CPR

- CPR is a full wrapper
- Graphics API fully hidden
  - not even #include
- Exposes similar primitives
  - Buffers, textures, render states, shaders, draw commands
- Statically linked C++
- Basically a new API of its own

```
//sample (from DOF code):

cpr::Render::setPrimitiveType( cpr::PRIMITIVE_TYPE_POINTS );
cpr::Render::setVertexLayout( &mDOFVertexLayout );

cpr::Render::pushBlendState( &mBlendAdd );
cpr::Render::setClearColor( 0.f, 0.f, 0.f, 0.f );

cpr::Render::clear( cpr::CLEAR_COLOR_BUFFER );
cpr::Render::setParam2f( mDOFParams.bokehSize, bs_far );
cpr::Render::drawIndexed( w*h );

cpr::Render::setFrameBuffer( &mDOFNearBuffer );
cpr::Render::clear( cpr::CLEAR_COLOR_BUFFER );
cpr::Render::setParam2f( mDOFParams.bokehSize, bs_near );
cpr::Render::drawIndexed( w*h );
```

CPR

- Interface must match common subset of backends
- Fortunately this subset is large
  - Mostly what you'd be working with anyway for cross-platform dev
- Opportunity to design our own interface to gfx
- Potential for all kinds of debugging, verification, etc.

CPR

- Which shader language to use?
  - GLSL?
  - HLSL?
  - NVIDIA’s Cg is dead :-/
- Exact language conversion is nontrivial
  - Existing solutions often don’t expose newest features
- Any compiler must work on all platforms
  - Precompilation not possible for us

CPR

- GLSL and HLSL are similar & have good preprocessors
- Insert macros to give both sets of key words
  - e.g. both float4 and vec4, lerp and mix, etc.
- Insert more macros to unify divergent syntax
  - e.g. main entry point, interpolants, texture samples, etc.
- Shaders written with these changes compile in both!
- Can always #ifdef platform-specific code too

CPR

Simple shader code syntax examples:

(please excuse the vertex shader)

```
//bloom (vertex shader)
uniform vec4    uPositions[4];
uniform vec4    uTexCoords[4];

BEGIN_PARAMS
    INPUT0(float,vID)
    OUTPUT0(vec2,fCoord)
END_PARAMS
{
    vec2 tcoord = uTexCoords[ int(vID) ].xy;
    #ifdef RENDERTARGET_Y_DOWN
        tcoord.y = 1.0 - tcoord.y;
    #endif
    fCoord = tcoord;
    OUT_POSITION = uPositions[ int(vID) ];
}
```

```
//bloom (pixel shader)
USE_TEXTURE2D(tInput);

uniform vec4    uKernel[BLOOM_SAMPLES];

BEGIN_PARAMS
    INPUT0(vec2,fCoord)
    OUTPUT_COLOR0(vec4)
END_PARAMS
{
    vec4 c = vec4(0.0, 0.0, 0.0, 0.0);
    HINT_UNROLL
    for( int i=0; i<BLOOM_SAMPLES; ++i )
    {
        vec3 k = uKernel[i].xyz;
        c += texture2D( tInput, fCoord + k.xy ) * k.z;
    }

    OUT_COLOR0 = c;
}
```

Image-Based Lighting

“Chinese Man” by Ballo | ballo.cghub.com

Image-Based Lighting

- Image-based lighting is key to image quality
  - The world is not made of point lights!
- Seen use in film for over a decade, more recently games
- HDR photography captures great real-world lighting data
- Light probes in game levels can do the same

Image-Based Lighting

- Usually done in real time with pre-convolved cube maps
  - Diffuse: store Lambert or similar (N dot L), look up with N
  - Specular: store Phong or similar (R dot L)^s, look up with R
  - Use mipmap chain for storing different Phong exponents
- Toolbag 1 used this approach, as have many others

Image-Based Lighting

- Looks pretty good!
- Has lengthy n^2 precomputation time :(
- Makes some BRDFs difficult or impossible
  - Blinn-Phong, Anisotropic
- Some shaders (e.g. skin) required custom convolutions

Image-Based Lighting

- GPU importance sampling
  - http://http.developer.nvidia.com/GPUGems3/gpugems3_ch20.html
- General idea:
  - Sample cube map several times to approximate integral
  - Sample directions distributed based on ‘importance’, or scale of BRDF
  - Can use mipmap LOD to simulate larger sample groups

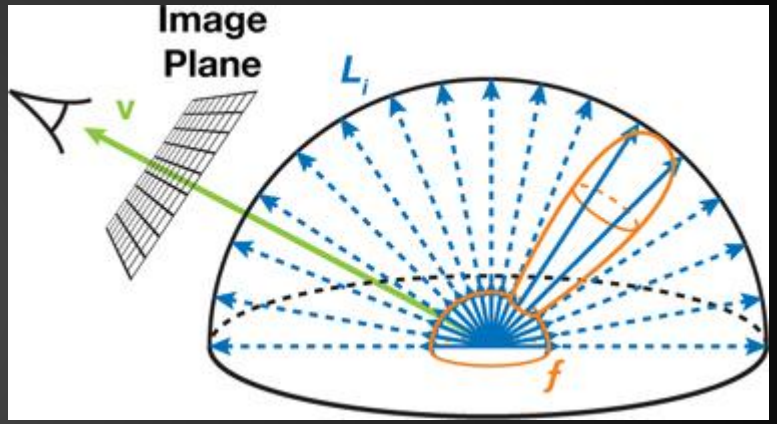


Image-Based Lighting

- Toolbag 2 has a few specular functions to pick from:
  - Simple Mirror (1 sample, basic reflection)
  - Blinn-Phong (32 samples, importance distribution)
  - Anisotropic (32 samples, ‘distorted’ importance dist. + aniso filtering)
- Can share the same cube map with ordinary mips
- No precomputation time
- Fast image import!
- Can do any BRDF!
- Cube seams can be an issue
  - Pre-process these edges if hardware doesn’t filter properly

Screen-space Ray Tracing

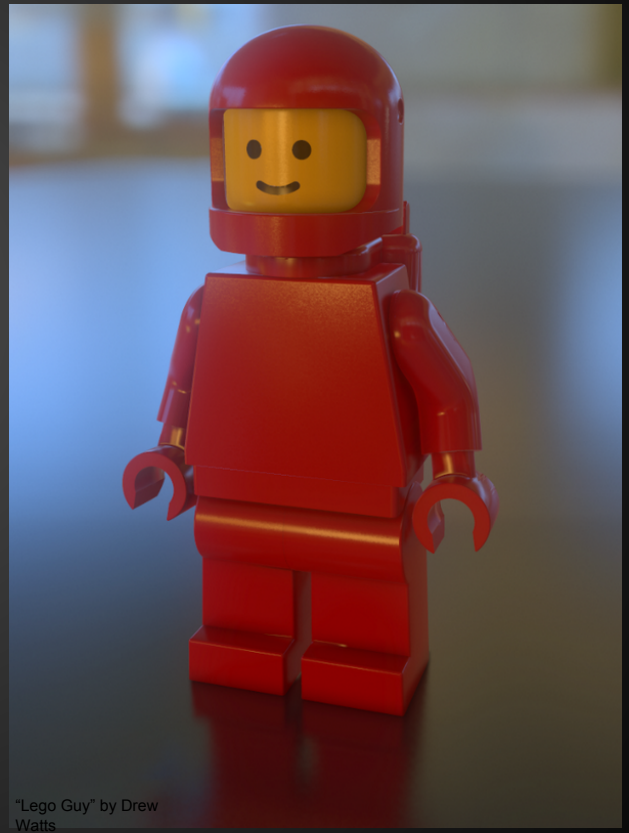
- IBL is great, mostly for spaces with distant backgrounds
- “Indoor” probes can be used, with parallax correction, but:
  - Objects that share a probe won’t reflect each other
  - Contact reflections are not present
  - Often results in a “floaty” look, especially with dynamic scenes
  - Cross-fade between probes looks funny
  - :(

Screen-space Ray Tracing

- Solution: just add ray tracing! Easy, right?
- k-d tree & full tracing etc. are too much right now
- Already have a sort-of-voxelized copy of the scene
  - The depth buffer!
- What if we just marched rays against that?
- Won’t be perfect, but maybe still useful if combined with IBL

Screen-space Ray Tracing

- Turns out this works pretty well!
- Good contact reflections
- Speed proportional to screen fill, not geometry
- Speed is usable for higher end GPUs



"Lego Guy" by Drew Watts

Screen-space Ray Tracing

Screen-space Ray Tracing

Screen-space Ray Tracing

- Performance Optimizations:
  - Create a course R32G32 min/max buffer from depth
  - Trace with big steps against this at a high level (limit 24 samples)
  - On intersection, perform (10) samples of full-res scene depth
  - On *that* intersection, perform (6) additional samples to refine position
- Best case (no hit): < 24 samples
- Average case: ~70-90 samples
- Worst case: 270 samples

Screen-space Ray Tracing

- Artifacts:
  - “Shadowing” occurs often where information is missing
  - Can’t do much but count it as a ray miss
  - This creates gaps in the reflected image
  - Can hide some by fading out rays pointing near the camera
  - Screen boundary is the other obvious limitation
  - Fade out as ray nears screen edges

Screen-space Ray Tracing

- Compositing:
  - No lit surfaces to reflect (haven’t done forward pass yet!)
  - We have the previous frame, but thats not very reliable data
    - Looks *really* bad for reflections to drop out or lag
  - Just trace against depth, and store resulting hit coordinates
  - After main pass, gather reflected light from color buffer and composite
  - Should *replace* IBL/environment lighting, not add!
  - This means the main pass must mask its specular with “hit mask”

Screen-space Ray Tracing

- BRDF interaction:
  - What if the surface is not perfectly smooth?
  - Reflection should blur
  - Remember our IBL importance sampling? We can do the same here
  - Screen-aligned noise texture, combined with gloss value can generate a random ray with importance distribution
  - With supersampling will approximate integral
  - Can also average with neighbors, but too much of this looks bad
  - Still trying to strike a good balance with the filtering on this

Supersampling

Supersampling

- MSAA
  - Good hardware support
  - Works well for edges
  - But not all edges (e.g. discarded pixels)
  - And not anything else (e.g. texture samples, reflections)
  - Can complicate compositing with multiple passes / effects

Supersampling

- FXAA (or similar)
  - Works on all pixels
  - No issues with compositing (all happens in post)
  - Fast!
  - Kind of a blurry look
  - Doesn’t help shader aliasing very well

Supersampling

- Supersampling
  - Works on everything
  - No issues with compositing
  - Very high quality
  - Super slow :(
- This is pretty much perfect for offline screenshots
- But not our viewport. Can we speed it up?

Supersampling

- Temporal Supersampling
  - Take the average over several frames
  - Use changing sub-pixel projection shifts
  - Requires rolling frame buffer history
  - Any number of samples you like
  - Very fast!

Supersampling

- Temporal Supersampling
  - Movement generates a motion-blur effect :(
  - Can partly fix with reprojection into prior frames’ projection/cam matrix
  - Can partly fix with velocity estimation
  - Still not great results
  - Easier for our purposes just to turn off the AA during certain operations, e.g. camera motion

Supersampling

- We use 4x temporal supersampling in Toolbag 2's viewport
- Can use just about any count during final render
  - 25 (5x5 kernel) is usually enough
- Sub-pixel shift can also be used for very large shots
  - Hardware has VRAM / implementation limits on output size
  - Composite multiple renders into large CPU-side buffer
  - Yields huge screenshots with no divided-frustum artifacts
  - Combines with supersampling too
    - e.g. 4x enlargement, 25x sampling = 100 renders

Supersampling

- Also allows for cool quality/speed optimizations
- Can get more “free” samples of:
  - Specular
  - Traced reflections
  - “Screen door” transparency (order independent!)
  - Shadows
  - Textures
- Add a mipmap LOD bias, or even turn it off
  - Supersampling covers texture filtering better than mipmapping does

Wrapping up

- Toolbag is a lot of fun to work on
- Fun playground for lots of graphics techniques :)
- We have great users who make smart feature requests
  - Shader source is open and editable in any installed copy!
- We plan to keep expanding the tools while we have an audience

Time for questions!

Thanks

Jeff Russell (@j3ffdr) - engineering

Andres Reinot (@monkeyscience8) - engineering

Mark Doeden (@markdoeden) - biz, ops, wearer of hats

Joe Wilson (@JoeWilsonEQ) - artist in residence, publicity