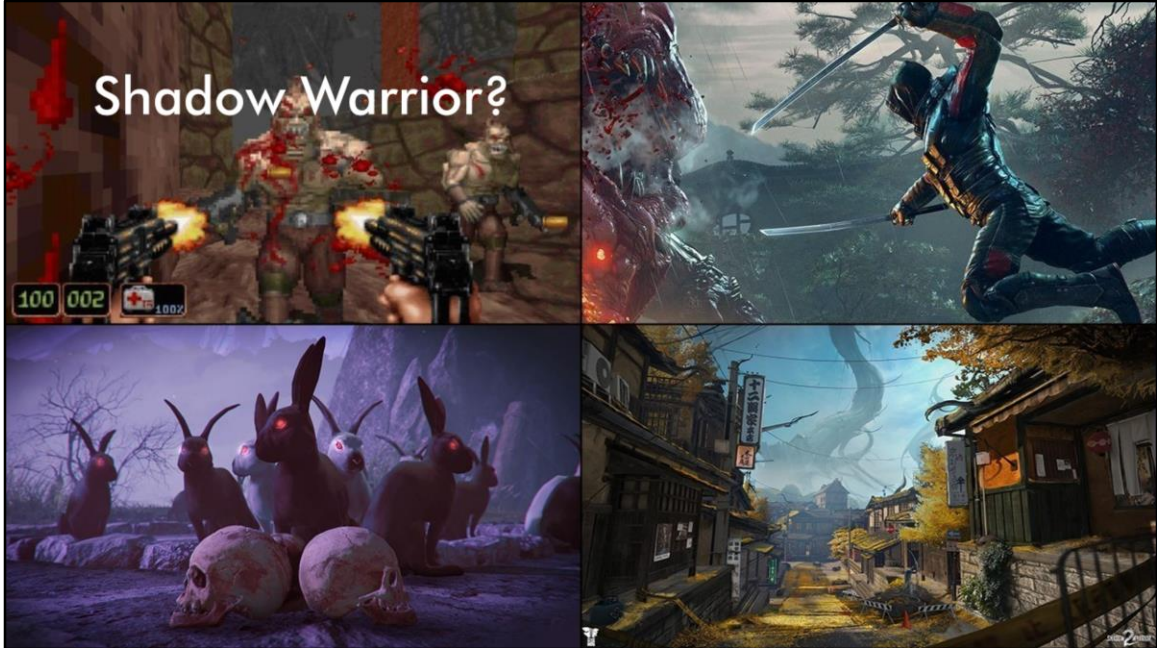


Hi, my name is Krzysztof Narkowicz. I'm the Lead Engine Programmer at Flying Wild Hog – a game development company located in Warsaw, Poland. Today I'm going to talk about rendering in our latest game called Shadow Warrior 2. Mainly, how our new procedural generation pipeline influenced our rendering.

*(Screenshot by "K putt" [https://www.flickr.com/photos/k\\_putt/29727530404/](https://www.flickr.com/photos/k_putt/29727530404/))*



What's this Shadow Warrior? It's a game based on a classic first person shooter from the '90s. It's most distinct feature is its focus on the melee combat. It's not a very serious game. It's more like an over the top one. For the art direction we strive for vivid colors and Japanese vibe.

# Shadow Warrior 2

Shadow Warrior (reboot) 2013	Shadow Warrior 2 2016	Increase
Corridor shooter	Looter shooter	
10 weapons	70 weapons	x7
10 enemies	50 enemies	x5
Small linear levels	Big open levels	x?
No multiplayer	4 player Co-op	∞
Kyoto	Kyoto/Mountains/Cyber	x3
Mid graphics quality	Top graphics quality	x?
40 people / 2 years	60 people / 3 years	x2



Back in 2013 we finished the first Shadow Warrior (reboot) - a corridor shooter style game and we decided to change to a lot for the second game. First of all, we wanted to make a looter shooter style game with larger open levels and vertical gameplay - we wanted to incorporate parkour (e.g. be able to jump onto the rooftops and fight enemies below). Additionally, we wanted to make tons of content – instead of 10 weapons as in the first game, we wanted to do 70. Instead of 10 enemies we wanted to do 50. We wanted to add multiplayer, 3 distinct art settings, substantially raise the graphics quality bar...

Basically, we wanted to add tons of content, but we didn't want to scale our team accordingly. I guess all of you know that when you start to scale your team too fast then all kinds of bad things can happen.

It was obvious we can't ship the second game using the same pipeline as we used for the first one.

# Procedural Generation To The Rescue

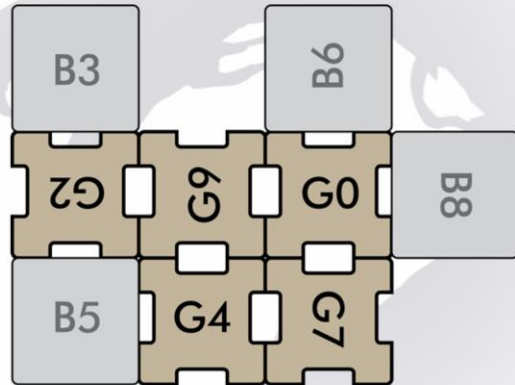
- Credit to the gameplay team
- Let's generate levels from handmade parts (at runtime)



Here comes the procedural generation to the rescue. Our gameplay team had this idea that let's make some parts of the level manually and automatically generate levels from those parts. We also wanted to do that at runtime (during the loading) in order to increase replayability.

# Blocks

- 200m by 200m
- Randomly rotated and placed
- Background blocks (vistas)



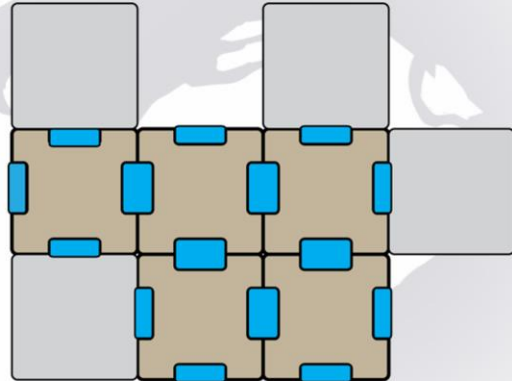
Let's take a look at our procedural level generation pipeline. We start from 200m by 200m square parts of the level, which we call blocks. We randomly rotate and connect blocks together in order to generate a level.

There are two kinds of blocks – gameplay blocks and background blocks. Gameplay blocks are basically where the player can move. Background blocks are used for distant forests, mountains etc. Basically, all kinds of vistas.

There is one more problem. We need some smart way to seamlessly connect two different blocks - we can't just end every block with a flat edge and a single global texture.

# Connectors and Blockers

- **Connectors**
  - 30m by 70m
  - Simplify variations
- **Blockers**
  - 15m by 70m
  - Shut off unused entrances

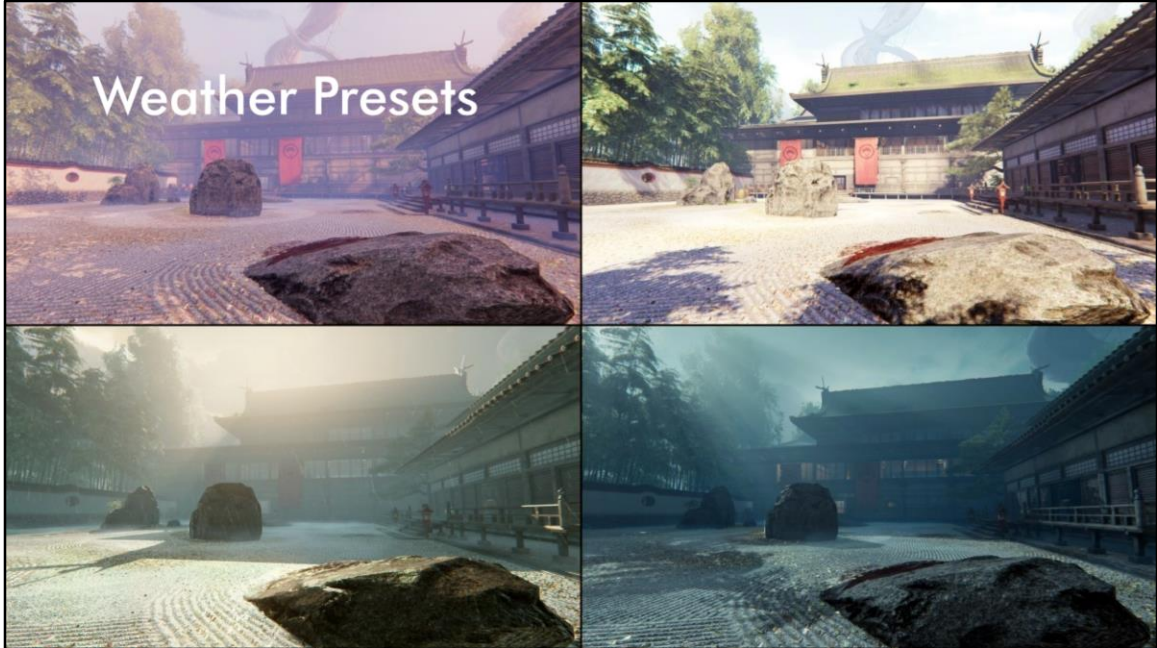


In order to connect two different block together we use connectors (blue pieces on the diagram). They are inserted into the special cutouts in the gameplay blocks and usually look like some kinds of gates, teleports, entrances or holes in walls.

We also have smaller connectors called blockers, which are used to shut of the unused entrances to our levels.



Layer presets allow for some randomization inside blocks. Four screenshots above are from the same point on the same block, but with different random layers enabled. Using layers we can for example replace a bridge with a destroyed bridge, replace river with a park or with buildings. Add some trees, randomize contents of the buildings, shut off some building entrances...



Weather presets make the final part of the procedural level generation pipeline. Basically, they are a set of lighting, post processing and weather settings.

All parts of our procedural generation pipeline are driven by scripted rules, which control which blocks and presets can be used for a given level.

That's all for the theory.



Obviously in practice it wasn't such a smooth ride. Especially our QA hated this new pipeline, as now they had to test tons of level variations. Additionally, level designers and artists lost manual control over the specific scenes.

# Rendering Challenges

- Dynamic lighting and geometry
  - Can't bake lighting, GI, AO, env maps, PRT...
- Procedural levels
  - Can't tweak exposure or lighting for a specific scene



Digital  
Dragons

FF

Procedural level generation pipeline strongly influences rendering.

Primarily, now we have dynamic lighting and dynamic geometry, which breaks a lot of our standard algorithms. We can't bake indirect lighting, AO, env maps or use PRT and we need to find a replacement for those algorithms.

Moreover, lighting artists can't tweak (hack) lighting or exposure for a specific scene, so rendering needs to deliver robust results without handholding.

We had to solve all those challenges in order to be able to ship this game.

## Indirect Lighting (GI)

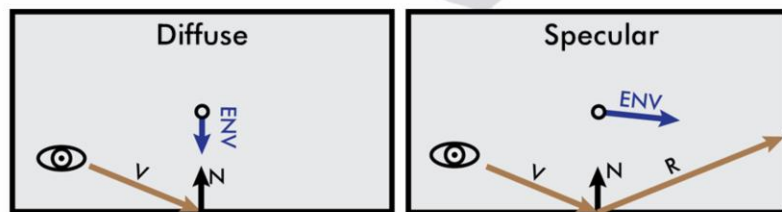
- Runtime generated env maps
- Placed inside prefabs
- Indirect diffuse and specular



Let's start from the indirect diffuse and specular. We place localized env maps inside prefabs, so when procedural generation places e.g. a building prefab it also adds some env maps to the scene. Next, we generate env maps at runtime and use them for sampling indirect diffuse and specular.

## Local Env Maps

- Custom shapes (convex...)
- Custom capture point and falloffs
- Prefilter with firefly filtering [Kar13]
- Last mip (8x8) – irradiance



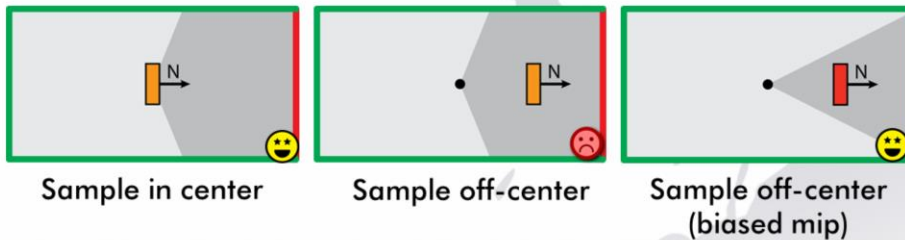
This obviously puts a lot of pressure on the quality of our localized env maps, so we spent substantial amount of time extending them. We have different shapes: box, ellipsoid and convex. Convex was the most popular shape. It's implemented as a ray trace against a set of planes and picking the nearest intersection point. We can set a custom falloff per every side of a proxy shape. We also allow to set a custom capture point, as usually you want the capture point to be at height of the player (~2m above ground) in order to maximize the angular resolution.

We prefilter our env maps using importance sampling at runtime, so we can't use too many samples. This results in some undersampling artifacts called fireflies (single very bright pixels). We remove fireflies using a simple firefly filter. In the last mip map level (8x8 pixels) we store irradiance (indirect diffuse).

As you can see on the diagrams, sampling is quite simple. We just shoot a ray against the proxy shape and sample the environment accordingly. Nice property of this solution is that we sample indirect diffuse and indirect specular from the same source, so they always match each other perfectly.

# Mip Bias

- Single convolution point
- Bias sample mip level (specular, diffuse) [LR14]



There is one additional issue. We prefilter only for a single point of capture, so the further you move away from it, the more incorrect lighting becomes. To fix that, we bias mip level a bit during sampling, as higher mipmap levels contain narrower filters.

# Real-Time Generation

- Too slow to generate during level loading
- Custom geometry pass
  - Disable dynamic objects, normal maps, detail maps...
  - Low LOD
  - Low res textures
- Amortized over 7 frames
  - 1st frame – shared shadow map
  - Next 6 frames – 6 env map faces
  - Last frame – filter and compress



Initial approach was to generate all the env maps during level loading (generation), but it was just too slow. First of all, level artists started to use env maps as a poor man's lightmapper and started to place tons of those. Moreover, in order to be able to generate all the env maps for a large level you need to stream a part of the level, generate, unstream it, stream another part etc. Basically, this resulted in unacceptably long loading times. We decided to generate env maps on the fly, as the camera moves through the level.

In order to do that efficiently, we have a special very fast geometry pass. It doesn't draw any kind of dynamic geometry or effects. It uses simplified shaders – without normal maps, detail maps etc. It also uses low LOD settings and low resolution textures, so there is no extra pressure on our texture streaming system. Low resolution textures were crucial for fixing stuttering on PC (resulting from VRAM oversubscription during generation), as on PC VRAM is shared with other applications (e.g. web browsers can easily grab 0.5-1gb of VRAM) and there is no way to manually control VRAM residency or usage.

Generation is amortized over 7 frames. On the first frame we draw a huge shadow map covering all the 6 views from 6 env map faces. During the next 6 frames we draw one env map face per frame. Together with the last face (pointing up) we do prefiltering and compression.

## Env Map Cache

- Real-time BC6H compressor in shader [Nar16]
  - 4mb -> 0.5mb per env map
- 128 cubemap array cache (6x256x256 BC6H)
  - Blend in/out
- < 1ms GPU load on XB1/PS4



For the compression we use a real-time BC6H compressor. It's implemented as a single pixel shader outputting BC6H compressed blocks and allows us to lower memory usage from 4mb to just a 0.5mb per env map.

After compression, we store our new env map inside a small cache – 128 element 256x256 cubemap array (it's a power of two, as some GPUs sneakily pad cubemap arrays to the next power of two). Next, we blend in the new env map. If we want to remove an env map from the cache, we first blend it out. This way there is no visible popping on screen during env map generation or discarding.

Entire system is quite fast - below 1ms of GPU load on consoles in the worst case, which is acceptable for a 30hz title. Of course, most of the time all env maps are already in cache and we don't have to generate anything.

## It Works



Env maps off



Env maps on

Obviously, our solution doesn't deliver the quality of properly baked lightmaps, but it works and provides coherent indirect diffuse and indirect specular lighting.

# Ambient Occlusion

- Micro scale – cavity from textures
- Macro scale - SSAO is ok (kind of)
- What about larger scale?



What about ambient occlusion? For the micro scale we have cavity stored in our textures. For the macro scale we have SSAO, which is kind of ok (better than nothing). But what about larger scale? What about buildings? Dynamic characters?

## AO Volumes

- Analytic shapes attached to prefabs



AO off



AO on

For the building AO we have a similar system as for the indirect lighting. We place some analytic shapes inside our prefabs and then at runtime we generate AO from those shapes.

## Grounding Characters

- No manual light control by artists
- Levitating characters



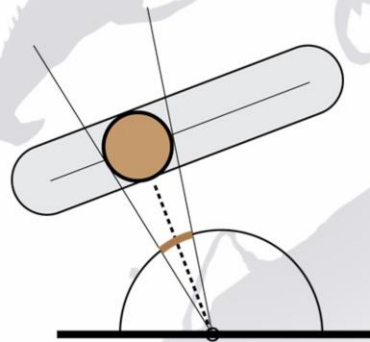
Digital  
Dragons

FF

What about dynamic characters? With procedural level generation our lighting artists can't tweak lighting for a specific scene, so often you may have situations like on this screenshot, where enemies stand in full shadow and don't cast any kind of shadow on their own. They look like if they are levitating above the ground.

# Indirect Capsule Shadows

- “Lighting Technology of TLoU” - Michał Iwanicki [Iwa13]
  - Aprox characters with ellipsoids
  - Trace cone vs ellipsoid on SPU
  - Lookup for occlusion
  - Cone dir from dir lightmap
- SPU -> compute shader
- Ellipsoid -> capsule
- Fake cone dir



Thankfully, Michał Iwanicki made a great presentation about the lighting in the “The Last of Us”. He had this nice idea to approximate characters using ellipsoids, trace cones against ellipsoids and compute indirect character shadows on the SPU. Additionally, “The Last of Us” had directional lightmaps, which provide the most important indirect lighting direction (cone direction) for every point on any surface.

Obviously, nowadays we don’t have SPUs, so we implemented everything using compute shaders. BTW most SPU tricks fit nicely with compute – e.g. triangle culling or particle rasterization.

Instead of ellipsoids we use capsules, which are handier for the graphic artists. BTW those aren’t real capsules, but more like those old-school line/tube lights. Namely, we place a sphere inside a capsule, move this sphere so it’s nearest to the evaluation point on the surface and then trace a cone against this sphere. Obviously, it’s much easier to do, than tracing a cone against a real capsule.

We also don’t have directional lightmaps (actually, we don’t have any lightmaps at all). Instead we derive a fake direction from the sun direction – we flip main directional light’s direction and force it to always point down (helps cases when there is no main directional light). This way we always have some indirect shadows on the ground and we don’t get any kind of double darkening of character shadows.

# Implementation

- 4 separate passes
- Tile depth bounds (shared)
- Coarse tile culling
  - Frustum AABB vs capsule bsphere
  - Bitpack results
- Tile occlusion
  - firstbitflow to unpack
  - 8x8 tile vs cone
  - Occlusion
- Bilateral upsample (shared)



Our implementation is separated into 4 passes. Splitting compute shaders into separate passes may seem counterintuitive, but often it leads to a superior performance, due to better VGPR usage or just being able to share computations between different algorithms.

First pass computes tile depth bounds and is shared among many algorithms (e.g. we also use tile bounds for volumetric lighting culling).

Second pass does coarse culling by testing tile frustum AABB against bounding spheres of capsules and bit packs results to a small render target.

Third pass unpacks previous results, culls marked 8x8 tiles against cones (computes occlusion for a worst case in the entire tile) and if a tile passes the test, then computes indirect shadows for this tile. Occlusion computation basically boils down to a single texture lookup.

Finally, we upscale our results from half res to full res inside a shared bilateral upsample pass (e.g. we upscale SSAO results together).

## Results



Capsule shadows off



Capsule shadows on



Indirect capsule shadows provide nice soft shadows and our characters are now properly grounded into the scene.

# Gore

- Important for melee combat
- SW1 – handmade
- Can't make 50 enemies!
- SW2 - mix of 3 systems:
  - Holes
  - Damage material layer
  - Cutting



As I mentioned in the beginning, melee combat is a core part of our game and for a good melee combat you want to have a good gore system.

In the previous games we were making gore manually. Graphics artists had to make lots of different chunks per enemy (see screenshot), create damage materials, script gore etc. For the Shadow Warrior 2 we planned to make 50 enemies, so there is no way, we could deliver that amount of content using the old pipeline.

In order to automate and simplify gore creation we made 3 new systems – holes, damage material layer and procedural cutting.

# Holes

- Inspired by L4D2 [Vla10]
- Vertex Shader
- Min dist to capsules/planes
- Export SV\_ClipDistance0
- Gun wounds, teleports...



Holes are inspired by the "Left 4 Dead 2" style gore. In vertex shader we compute a minimum distance to a set of planes and capsules. Then we use a single hardware clipping plane together with that distance to cut a hole inside character. Holes are later covered with some manually created chunks, which are reused between different holes.

This system was primarily intended for the gun wounds - like the one on the screenshot, where the enemy has this huge hole inside his chest. Of course, gameplay found more creative uses for it - e.g. they used that for teleports (two planes – one moving bottom up and the second one moving top to bottom) or for the enemy spawners (e.g. when enemies are digging out from ground, you don't want to render their part, which is below the ground).

## Damage Material Layer

- Ice, fire, acid, electricity...
- Shader `#ifdef`
- Vertex masked heightmap blend



Shadow Warrior 2 is a looter shooter style game, so we have different types of damage (ice, fire, acid etc.) and we need a way to automatically apply those damage effects to any enemy.

Every character shader has an additional top material layer hidden behind a shader define. Damage material layer is implemented as a standard vertex masked heightmap blend of damage material on top of character shader.

# Procedural Cutting

- Credit to:
  - Jarosław Pleskot
  - Jacek Miąskiewicz
- Used for:
  - Katana, chainsaw...
  - Gun wounds
  - Explosions

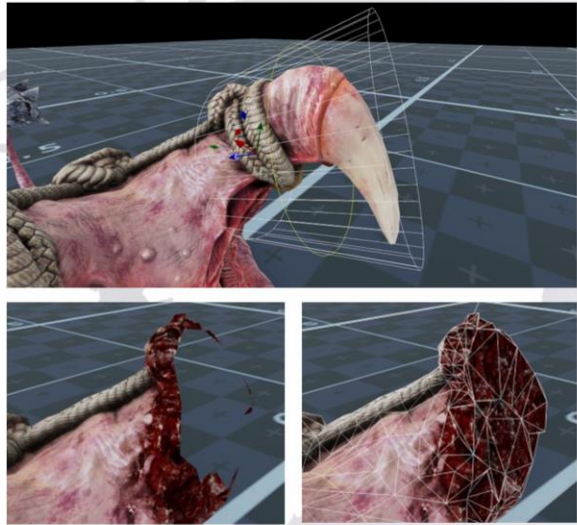


Finally, the coolest system – procedural cutting. Here the full credit goes to Jaroslaw Pleskot and Jacek Miaskiewicz – guys who developed it.

The main motivation behind procedural cutting was being able to cut the enemy mesh exactly where the katana, chainsaw (or a chainsaw-katana, which is actually a thing in our game) went through the body of the enemy. Additionally, it's used for automating gun wounds – e.g. to shoot off an arm. It's also used for explosions, where we cut enemies into small pieces and send them flying.

## Cutting 2

- Apply cut bounds
- Cut geometry in T-pose
- Add a "cap"
- Remove attachments



First we start from cut bounds (cylinder on the upper screenshot). Cut bounds allow gameplay to alter or disallow some cuts. For example, we don't want to cut a large enemy into two parts with the first blow - first player needs to wear him off.

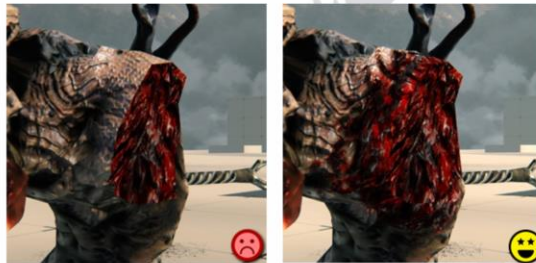
Next step is to cut the geometry. We separate geometry into two clean parts – one on the left side of the cutting plane and one on the right side. We cut in T-pose, as mesh geometry must be water-tight and after applying animation it can become self-intersecting. Plane's normal and origin is computed by transforming a single hit point from world space to T-pose space.

Now we have two cut pieces with holes (lower right screenshot). In order to cover holes we triangulate them. Additionally, we also tessellate holes, as we need extra vertices for proper skinning.

If there were some attachments near the cut (e.g. chains or some pieces of cloth), we detach them and drop them on the ground.

## Cutting 3

- Vertex unmask blood layer
- Large parts – reskin and cut the ragdoll
- Small parts – static mesh



Digital  
Dragons



Next step is to vertex unmask a blood layer around the cut in order to hide harsh transition between the cut and the rest of the character (see screenshots).

Finally, we classify new parts. If it's a small part like a finger then we remove skinning and spawn it as a static mesh. If it's a large part like rest of the body after cutting a finger, then we reskin new vertices using envelopes and cut the ragdoll.

## New Formats

- Texture - BC7
  - Enc normal (RG) + roughness (B)
  - Handy for Toksvig prefiltering
- Render Target - 11\_11\_10\_float
  - Replaces RGBA16\_float
  - Pre-expose and dither
  - Crazy good on XB1 (2x stuff in ESRAM)



For the Shadow Warrior 2 we started to use a lot of new (kind of) formats like BC7 for textures. BC7 is great e.g. for storing encoded normal maps together with roughness. It not only lowers memory usage and bandwidth, but also coupling of normal and roughness is handy for the Toksvig prefiltering.

We also entirely moved away from 64bit HDR render targets and now we use exclusively 32 bit HDR render targets. Specifically the 11\_11\_10\_float. This format provides surprisingly good quality if you pre-expose (pre-multiply all shader outputs by previous frame's exposure) and add some dithering. It's not a life changer on PC or PS4, but it's incredible on XB1. XB1 has a small amount of very fast ESRAM and now we can double the amount of render targets inside ESRAM, which really helps the performance.

# Layered Tiled Materials

- Want:
  - Blend 4 materials (art)
  - Do it fast! (code)
- 4 textures in an texture array
- 4 weights from vertex color / texture
- Blend 2 most important materials per pixel



We also started to use a lot of layered tiled materials. Graphic artists love layered tiled materials, but most programmer don't share that love, as they often can be quite slow to render.

We place our tiled materials inside texture arrays, read some weights form a vertex color or from a separate mask texture. Next, inside pixel shader we pick two most important materials per pixel, renormalize the weights and blend the selected materials.

This approach requires manual seam correction in places where more than two materials connect. On the other hand the worst case is still very fast, as we never blend more than 2 materials. Overall, we feel that it was a good tradeoff, as it greatly simplifies material performance maintenance.

## Volume Decals

- Variety and hiding seams
- Custom shapes and falloffs
- Proj decoupled from falloff dir
- Custom output modes
- Heightmap based blending
- World space noise mask



Our graphic artists love volume (deferred) decals. They use them for adding some variety and for hiding seams between the objects. We extended our volume decals with tons of different options:

- Custom shapes. E.g. we have a cylinder which was used e.g. for spawning fallen leaves around trees.
- Custom falloffs. E.g. angle falloff which is decoupled from the projection falloff. This way it's possible to project snow top down, but setup the angle falloff from the right side, so the snow will cover only the right side of the object.
- Custom output. We can override or blend existing normals, output only specified properties (e.g. just emissive) or output weather properties like wetness, which is later used to spawn puddles.
- Different blending modes – the one mostly used was a heightmap blend – very handy for things like moss, where you want sharp transitions.
- On top of all that we have a world space noise masks, which add extra variety inside decals. It's especially handy for placing large decals and still having some variety inside of them.

# Automatic Exposure

- Want: exposure = 1 / dominant lighting [Hof13]
- Easy to use EV units (camera stops) [Ree14]
- Log2 space histogram



Expose for interior



Expose for average



Expose for exterior



Let's go back to one of our main challenges - automatic exposure. What to expose for, when lighting artists can't tweak exposure for a specific scene due to the procedural level generation? Let's look at the photos below. Here we have three options:

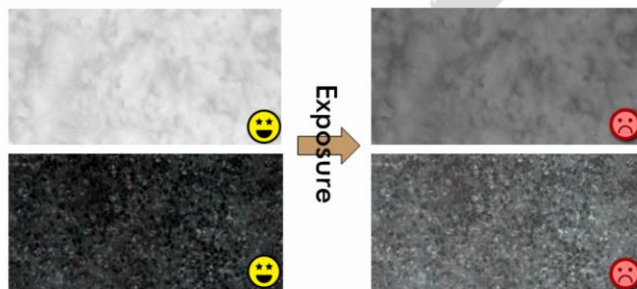
- Expose for the interior - interior is well exposed, but most of the scene (exterior) is almost plain white.
- Expose for the average (standard exposure algorithm) - interior is too dark and exterior too bright and misses many important details.
- Expose for the exterior - interior is too dark, but on the other hand most of the scene is well exposed and keeps all the important details. This is the idea behind our automatic exposure – try to find the dominant lighting condition on the screen and expose for that.

It plays really nicely with our lighting units – EVs. For example, if our sunlight is set to 16, then a good starting point for exposure is also 16. We obviously don't set exposure manually, but it helps lighting artists to understand the exposure debug view and exposure controls. Instead of doing standard (weighted) screen space illuminance average, we use a histogram computed inside a compute shader, skip outliers and try to expose for the dominant lighting condition on the screen.

(Photos from <https://fstoppers.com/strobe-light/hdr-vs-flash-interiors-and-real-estate-photography-3135>)

## Automatic Exposure - Illuminance

- Want: same exposure for dark and white albedo
- Expose to lighting illuminance [Hof13]



Digital  
Dragons

3D

Automatic exposure can't distinguish between a dark surface and dark lighting. For example, let's look at white snow (upper left screenshot). Automatic exposure will decide that lighting is too bright, decrease exposure and you will get gray mess instead of the white snow (upper right screenshot). With the same lighting conditions and dark marble (lower right screenshot), automatic exposure will decide that lighting is too dark, increase exposure and again you will get gray mess (lower right screenshot).

In order to fix that, we expose for the lighting illuminance instead of final pixel illuminance. Basically, we expose using our diffuse lighting render target (we treat every surface as a fully diffuse and fully white surface). This way automatic exposure doesn't depend on the albedo, snow stays white and dark marble stays dark.

*(Images from maxTextures.com)*

# Exposure Compensation Curve

- Want: darker cavern than outside [KMS05]
- Exposure compensation curve
- Map current EV to exposure bias



We want to have a brighter image as average scene illuminance grows (e.g. we want to have a brighter image during a day than at night or in a dark cavern). Automatic exposure doesn't care about that and just tries to maintain a constant screen brightness.

In order to solve this issue, we use an exposure compensation curve. Basically, we allow our lighting artists to map current metered EVs to a custom exposure bias.

*(Photo source: Mike Cooper - "District 34")*

## Exposure VS FX

- Direct sunlight ( $\sim 100000$  LUX)
- Full moon lighting ( $\sim 0.25$  LUX).
- Want: maintain  $\sim$ constant screen brightness
- $\text{color} = \text{exp2}(\text{lerp}(\text{log2Color}, \text{log2Color} / \text{exposure}, \text{ignoreExposure}))$
- Just don't use the output for the auto exposure!



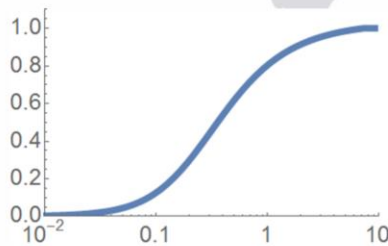
Finally, there is a problem of exposure versus FX and especially versus gameplay stuff.

In a game we have enormous ranges of luminance values for lighting – from 100k LUX direct sunlight to 0.25 LUX for full moon lighting. Obviously, gameplay doesn't care about our PBR mumbo jumbo and just wants their magic energy beams to have (almost) constant brightness on the screen.

We solve that with a simple hack, which allows artists to blend between a normal color and a color divided by the exposure. Effectively, allowing to smoothly choose between a normal color and a constant screen color. Just watch out and try not to use hacked illuminance values for the automatic exposure, as it introduces a feedback loop. One way to do that is to compute automatic exposure right after the main opaque geometry pass.

# ACES Approximation

- Started with a custom “filmic” curve
  - $(x*(a*x+b))/(x*(c*x+d)+e)$
- Simple ACES luminance only curve fit
  - Code: <https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/>



We started this project using a fully custom tone mapping curve. Near the end of the project everyone started to talk about ACES and we decided to check it out. We really liked the ACES tonemapping curve (RRT), but it was also quite complicated and slow. Instead of implementing the full ACES, we just fitted a simple analytic curve to ACES and used that as our tonemapping operator.

# Credits

- Thanks to our amazing team!
- Thanks to Paweł Dobosz for the help with the presentation layout!





*(Screenshot by "Sk)yline / v.2"*

*<https://www.flickr.com/photos/144479468@N08/33815356532/>*

## References

- [Kar13] Brian Karis - "Tone mapping", 2013, <http://graphicrants.blogspot.com/2013/12/tonemapping.html>
- [LR14] "Moving Frostbite to Physically Based Rendering" - Sébastien Lagarde, Charles de Rousiers, Siggraph 2014
- [Nar16] Krzysztof Narkowicz – "Real-time BC6H Compression on GPU", GDC, 2016
- [Iwa13] Michał Iwanicki - "Lighting Technology of "The Last Of Us"", Siggraph, 2013
- [Vla10] Alex Vlachos – "Rendering Wounds in Left 4 Dead 2", GDC, 2010
- [Hof13] Naty Hoffman – "Outside the Echo Chamber: Learning from Other Disciplines", i3D 2013
- [Ree14] Nathan Reed – "Artist-Friendly HDR With Exposure Values", 2014, <http://reedbeta.com/blog/artist-friendly-hdr-with-exposure-values/>
- [KMS05] Grzegorz Krawczyk, Karol Myszkowski, Hans-Peter Seidel – "Perceptual Effects in Real-time Tone Mapping", SCCG 2005