
SIGGRAPH2012
The 39th International Conference and Exhibition
on Computer Graphics and Interactive Techniques

Creating Vast Game Worlds

Experiences from Avalanche Studios

Emil Persson
Senior Graphics Programmer

@_Humus_

AVALANCHE STUDIOS

SIGGRAPH2012

Just how big is Just Cause 2?

- Unit is meters
  - [-16384 .. 16384]
- 32km x 32km
  - 1024 km^2
  - 400 mi^2

Just Cause 2 is a very big game. Not necessarily the biggest of all times, but certainly one of the bigger titles out there. Size is not necessarily an easy metric to compare either, considering that a space game would by default beat pretty much everything else. It is more of a scale vs. density ratio, and whether your content is artist created or procedural. Our content is almost entirely artist created. This talk is not about comparing though, but addressing the issues you may run into when creating very large game worlds.

Issues

- The “jitter bug”
  - Vertex snapping
  - Jerky animations
- Z-fighting
- Shadows
  - Range
  - Glitches
- Data
  - Disc space
  - Memory
- Performance
  - View distance
  - Occlusion

There are also game-play considerations, but we will not focus on that.

Breathing life into the world

This screenshot illustrates a few of the things we did to give the world more life. The distant lights system highlights all the locations in the world in a very natural way and makes the world look much more realistic. But most important of all, the world looks inhabited. It also helps the player find interesting points to go to and recognize areas of interest. The effect is of course more pronounced at night, but we have it enabled during day-time also.

The other interesting point here is our landmark system. For performance and memory reasons we can never have the whole world loaded, so only the relatively close locations are in memory. But for key landmarks and locations we have simple landmark models resident at all times. For instance, over the right shoulder of Rico in this image you see the twin tower casino from the second agency mission. The location is many miles away and certainly not loaded at this distance.

Breathing life into the world

- Landmarks
- Distant lights
- World simulation
- Dynamic weather system
- Day-night cycle
- Diverse game and climate zones
  - City, arctic, jungle, desert, ocean, mountains etc.
- Verticality

World simulation – Makes sure that there is always something going on. It could be AI planes flying over your head, civilians, birds, butterflies, scorpions etc.

Dynamic weather system – It could start to rain or snow at any time, even thunder. Also sets the right mood for some certain missions.

Day-night cycle – Adds diverse lighting conditions and makes the world feel more dynamic and living.

There are plenty of different zone that give the world a more diverse appearance and makes it more interesting. It also gives the player another reason to explore different areas.

We spent a lot of time making verticality work, because in this game you will spend plenty of time up in the air, whether in a helicopter or plane or simply slingshotting around with the grappling hook and parachute.

Distant lights

- Static light sources
  - Point-sprites
  - Fades to light source close up
- Huge visual impact
- Cheap

At compile time we assemble all the static light sources into a compact vertex buffer. We do some simple splitting of it into a grid for basic frustum culling and better accuracy for the packed vertex positions. The whole thing is rendered as simple colored point-sprites and is very cheap, about 0.1-0.2ms on the consoles on a typical frame, and has a huge visual impact. It was really a wow-moment when it first entered the build.

Distant lights

Another screenshot illustrating the impact of the distant lights and landmark system. The city really looks like a living city at many miles away, and with the city actually not being loaded into memory at all. The landmarks of the skyscrapers gives the city its spatial profile and the light sources give it its lighting profile and makes it seem alive. The keen eyed viewer may notice that bridges are missing, they are not loaded at this distance, but you still get an impression of them being there due to the long lines of regularly spaced light sources there.

Floating point math

SIGGRAPH2012

- floats abstract real numbers
  - Works as intended in 99% of the cases
    - Breaks spectacularly for 0.9%
    - Breaks subtly for 0.1%
    - *"Tricks With the Floating-Point Format"* Dawson, 2011. [6]
  - Find the bug:

```
// Convert float in [0, 1] to 24bit fixed point and add stencil bits
uint32 fixed_zs = (uint32(16777215.0f * depth + 0.5f) << 8) | stencil;
```

  - Logarithmic distribution
    - Reset FP timers / counters on opportunity
    - Fixed point

The bug in the code is that if you pass depth = 1.0f, the depth bits in fixed_zs will be 0, rather than the expected 0xFFFFFF. The reason for this is a combination of lack of precision and IEEE-754 rounding rules. In the upper half of the range here, i.e. from 8M to 16M, a floating point value has integer resolution. So there is no 16777215.5f value, only 16777215.0f and 16777216.0f. What happens when 0.5f is added to 16777215.0f is that the result is rounded to the closest infinitely precise value, and in cases where there is a tie, such as is happening here, it rounds to the closest even. This may seem weird, but is to avoid a rounding bias. What happens thus in our case is that we end up with 16777216.0f, or 0x1000000, which after the shift loses its left-most 1 and only zeros remain. This is just one example of where the abstraction of real numbers becomes leaky. It is of great importance to be aware of the underlying representation and its limitations and quirks.

The most basic insight about floats is that they have a close to logarithmic distribution of values. The greater the magnitude, the lower the resolution. For monotonically increasing values, something could work initially, only to fail after enough time has passed. For instance, animation gets jerky after some amount of time. Among other things, this happened for our water animation. The solution was to reset the timer as soon as no water patch was visible, and otherwise we also reset it after a few hours straight. This happened occasionally if someone left the office with the game running. The next day the water animation was extremely jerky.

Whenever it made sense, we switched to a fixed point timer, which does not suffer from this problem.

Floating point math

- Worst precision in ±[8k, 16k) range
  - That's 75% of the map …
  - Millimeter resolution
- Floating point arithmetic
  - Error accumulating at every op
    - More math ⇒ bigger error
    - add/sub worse than mul/div
  - Shorten computation chains
    - Faster AND more precise!
    - Minimize magnitude ratio in add/sub

Range	Increment
[8, 16)	1/1M
[8k, 16k)	1/1k
[8M, 16M)	1
[8G, 16G)	1k

For us the worst precision on world coordinates was in the 8k to 16k range. On 75% of the map, either x or z coordinate will be in this range, so it's essentially a universal problem. Floats have a millimeter resolution at this range, so it is actually OK if it was only used as a final stored value, but we also need to do math, doing transforms and all, resulting in accumulated error in every operation. So millimeters can turn into centimeters or even decimeters depending on how much math you do, and most importantly, how you do it.

All basic floating point operations introduce up to 0.5 ULP error for every operation. This is problematic in itself; however, by far the biggest offender is adding or subtracting numbers of very different magnitude. If a big and small number is added together the resulting float may not have enough resolution to properly represent the small number's contribution. Adding a number of much greater magnitude to a number, and then subtracting another big number from it, is a quick way to destroy any sort of precision the original value might have had.

Transform chains

SIGGRAPH2012

- Don't:

```
float4 world_pos = mul(In.Position, World);
Out.Position = mul(world_pos, ViewProj);
```

- Do:

```
float4 world_pos = mul(In.Position, World);
Out.Position = mul(In.Position, WorldViewProj);
```

- Alternatively:

```
float4 local_pos = mul(In.Position, LocalWorld);
Out.Position = mul(local_pos, LocalViewProj);
```

$$[W] \cdot [V] \cdot [P] =$$
$$[R_w \cdot T_w] \cdot [T_v \cdot R_v] \cdot [P] =$$
$$[R_w \cdot (T_w \cdot T_v)] \cdot [R_v \cdot P]$$

For the sake of the discussion here, let's assume the shader also needs the world_pos value, not just the output position.

First method is what we used early in development of Just Cause 2. It was very lossy because we deal with large world_pos values compared to the relatively small values in the input vertex position.

We switched to the second method which does not go through a world position, but transforms directly from input position to output. Note though that you can't just take the W, V and P matrices and slap them together, because you will run into similar floating point issues while computing this matrix. You need to decompose the matrices into their subcomponents and merge the translations from world and view matrices to take away the big translations.

The second method has the downside of requiring two per-instance matrices, resulting in more vertex shader constants to be set. In the first method the ViewProj matrix can be constant for the whole frame. Lately we have poked around with the third method. By centering the view-proj matrix around the camera we can keep a local view-proj matrix for the whole frame, and since we're working in local space we don't have any big translations. It will not be quite as precise as method two since we're chaining transforms again, but it is good enough in practice for us.

Never invert a matrix!

- Never invert a matrix! Really!
- Don't:

```
float4x4 view = ViewMatrix(pos, angles);
float4x4 proj = ProjectionMatrix(fov, ratio, near, far);
float4x4 view_proj = view * proj;
float4x4 view_proj_inv = InvertMatrix(view_proj);
```

- Do:

```
float4x4 view, view_inv, proj, proj_inv;
ViewMatrixAndInverse(&view, &view_inv, pos, angles);
ProjectionMatrixAndInverse(&proj, &proj_inv, fov, ratio, near, far);
float4x4 view_proj = view * proj;
float4x4 view_proj_inv = proj_inv * view_inv;
```

You often need to use inverse matrices. Inverting a matrix is often a quite lossy process. It is much more accurate to compute the inverse matrix directly from the same parameters that was used to generate the original matrix. It is usually way faster too.

How to compute an inverse directly

- Reverse transforms in reverse order
  - Rotate(angle) × Translate(pos) ⇒ Translate(-pos) × Rotate(-angle)
- Derive analytically from regular matrix
  - Gauss-Jordan elimination [1]

$$\left[\begin{array}{cccc|cccc} x & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & z_0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & z_1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \Rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & \frac{1}{x} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \frac{1}{y} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \frac{1}{z_1} \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & \frac{z_0}{z_1} \end{array}\right]$$

For a matrix built from standard transforms, such as a typical world or view matrix, the inverse is just the reverse transforms in the reverse order.

Some matrices, such as projection matrices, are less intuitive how to compute the inverse directly. A relatively straightforward approach is to use Gauss-Jordan elimination to derive the inverse analytically. Insert variables for the slots that your input parameters go into, do row-operations until you arrive at identity on the left and you have a symbolical solution on the right. Then plug in your original parameters into your variables and optimize order of operations to minimize precision loss.

Depth buffer precision

- Just Cause 2 has 50,000m view distance
  - That's all the way across the diagonal of the map!
  - Reversed depth buffer (near = 1, far = 0)
    - Helps even for fixed point depth buffers!
    - Flip with projection matrix, not viewport!
  - D24FS8 on consoles, D24S8 on PC
  - Dynamic near plane
    - Normally 0.5, close up 0.1

A standard solution is to reverse the range and use a floating point depth buffer, which we did. We actually shipped with 24bit fixed point buffer on PC, because DX10 does not have a D24F format that the consoles have, and we needed stencil. We could have gone with D32F_S8_X24, but since 24bit was enough we stuck with that.

It is a somewhat unintuitive result that reversing the depth buffer also helps with a fixed point depth buffer considering the resolution is uniform and there is a unique 32bit float matching every value in 24bit fixed point. The reason this is the case is that plenty of the error that can be introduced in the process happens before the rasterization, in other words, you get poor inputs to the rasterizer from the vertex shader and the lowest bits would for all practical purposes be noise. As we saw a few slides back, we are really operating on the edge of the precision of floats here when you need 24bit output, and any sort of math you do will likely lose a bit or two before you are done, even if we are careful about it. With a reversed projection matrix the math will mostly operate in a range that is much more comfortable for floats, i.e. close to zero, and the rasterizer is given more precise values as inputs. This is also why flipping the depth range should be done with the projection matrix, not with the viewport, because at that point it is already too late.

When comparing z-fighting prone cases with regular and reversed fixed point buffers, in practice the reverse has consistently resulted in better quality. It has a slight edge in the amount of saw-tooth artifacts, but the primary difference is the stability. In problematic areas the reversed fixed point buffer will still exhibit some artifacts, but they will be stable and thus less objectionable. A regular depth buffer will flicker in motion between about the same artifacts as the reversed to something far worse. A stable saw-tooth pattern in the distance may go unnoticed by a player unless it is pointed out, but a flickering mess will instantly catch the attention and is hard not to notice.

PS3 depth resolve

- HW has no native depth texture format
  - D16 can be aliased as an L16
  - D24S8 and D24FS8 aliased as RGBA8
    - Shader needs to decode
    - Lossy texture sampling
    - Beware of compiler flags, output precision, half-precision etc.

We had a lot of problems with the precision from our PS3 depth resolve shader and had to fix it numerous times. The hardware doesn't have any native depth formats, so it has to be resolved manually. Fortunately, the D24FS8 format lines up well with the channels when aliased as an RGBA8 texture. D24F is like a regular float, except it has no sign bit and fewer mantissa bits. The lack of a sign bit makes the exponent align nicely into an 8bit texture channel, which is nice. The amount of shader logic that needs to be done is actually not the much, but there are many pitfalls, both with the hardware and software pipeline.

PS3 depth resolve


```
#pragma optionNV(fastprecision off)

sampler2D DepthBuffer : register(s0);

void main(float2 TexCoord : TEXCOORD0, out float4 Depth : COLOR)
{
    half4 dc = h4tex2D(DepthBuffer, TexCoord);

    // Round to compensate for poor sampler precision.
    // Also bias the exponent before rounding.
    dc = floor(dc * 255.0h + half4(0.5h, 0.5h, 0.5h, 0.5h - 127.0h));

    float m = dc.r * (1.0f / 256.0f) + dc.g * (1.0f / 65536.0f);
    float e = exp2( float(dc.a) );

    Depth = m * e + e;
}
```

The first thing to ensure is that the flags passed to the shader compiler does not cause lower precision. We ended up having to override a global shader compiler flag for this shader by using a #pragma, because the fastprecision flag is generally beneficial for us for other shaders, but here it caused severe precision loss.

A key insight is that sampling textures is not particularly accurate. The hardware cheats a fair bit, so multiplying those values in [0,1] by 255 is not going to land you at integer values, just somewhere relatively close, but off by enough to introduce visible artifacts when the depth value is later used in shading. This is corrected by rounding to closest integer. We also move the exponent bias before the rounding to guarantee that it's an exact integer when passed to exp2() in the end. It turned out subtracting 127 from an integer oddly enough introduced error. Technically it shouldn't, even with halfs.

This final shader is both fast and accurate. For us it turned out ROP bound in the end and can thus be considered to be operating at best possible performance. The output is written to an R32F texture.

Shadows

- 3 cascade atlas
  - Cascades scaled with elevation
  - Visually tweaked ranges
- Shadow stabilization
  - Sub-pixel jitter compensation
  - Discrete resizing
- Size culling
- Xbox360 Memory ↔ GPU time tradeoff
  - 32bit → 16bit conversion
  - Memory export shader
  - Tiled-to-tiled

We are using 3 cascades in an atlas. This allows single-sample shadow fetches. To hide the seam, we dither between cascades.

The range of the buffer is scaled with elevation, so that we cover a larger area when you're up in the sky, like flying a helicopter etc. Actual ranges we use are tweaked visually together with artists to look the best in a number of real-world situations. This provided better results than the common "optimal" approach proposed in the literature with a logarithmic scale between cascades.

For shadow stabilization under movement we applied a standard jitter compensation, essentially snapping the texels to a grid. This sort of solution fails when the range of the shadow buffer is changed. This was solved by only resizing the ranges in discrete steps. For us to actually apply a change, the new requested buffer range has to differ by at least 7 percent from what's currently in use. On the resizes a trained eye may be able to detect a single frame snap of some shadow map texels on shadow edges, but if you don't know what to look for you will never notice. It is hard even for the trained eye.

For performance we cull objects smaller than a certain size, like for instance 3 pixels in the shadow map. This took away lots of small objects from the shadow map that didn't really contribute much. This way performance stayed relatively stable under resizing, such that e.g. a 50m range and a 200m range cost about the same to render. This was very nice because it took performance concerns off the table and allowed us to focus entirely on quality when tweaking ranges.

On Xbox360 we did a special memory optimization, because we had more problems with video memory there than PS3 but on the other hand more GPU time to spare. So we burned 1ms on a conversion from 32bit shadow maps to 16bit (because HW doesn't have any native 16bit format). Some of that we gained back due to cheaper sampling, so total cost may be in the range of 0.8ms or so. For this we saved 6MB of memory.

Memory optimization

- Temporal texture aliasing
  - Shadow-map, velocity buffer, post-effects temporaries etc.
  - Ping-ponging with EDRAM
- Channel textures
  - Luminance in a DXT1 channel
  - 1.33bpp
- Vertex packing

We analyzed the frame for overlap in use of render targets. For instance, the shadow map and post-effects temporaries were never used at the same time. So we aliased those on the same memory area on the consoles where we have full control over memory. For Xbox360 we also took advantage of that fact that whenever we are ping-ponging between render targets, we don't actually need a second render target, because the texture lives in video memory while the render target lives in EDRAM, so there is no read-after-write hazard.

For some relatively generic uniformly colored textures we optimized by packing several textures into the channels of a single DXT1, then configured the samplers to return a specific channel as a luminance texture. We added back some color using vertex colors. This resulted in very compact storage and worked well for a subset of our textures.

Vertex compression

- Example “fat” vertex

```
struct Vertex
{
    float3 Position;     // 12 bytes
    float2 TexCoord0;    //  8 bytes
    float2 TexCoord1;    //  8 bytes
    float3 Tangent;      // 12 bytes
    float3 Bitangent;    // 12 bytes
    float3 Normal;       // 12 bytes
    float4 Color;        // 16 bytes
};          // Total:    80 bytes, 7 attributes
```

This is a hypothetical fat vertex you may have at some initial prototyping phase that we will use for illustrative purposes.

Vertex compression

- Standard solutions applied:

```
struct Vertex
{
    float3 Position;     // 12 bytes
    float4 TexCoord;     // 16 bytes
    ubyte4 Tangent;      //  4 bytes, 1 unused
    ubyte4 Bitangent;    //  4 bytes, 1 unused
    ubyte4 Normal;       //  4 bytes, 1 unused
    ubyte4 Color;        //  4 bytes
};          // Total:     44 bytes, 6 attributes
```

Here we have applied a few standard solutions to our vertex. Most attributes didn't need the float precision and have limited range, so we store them as bytes instead. The two texture coordinates can also be merged into a single float4. It is now nearly half the size and one attribute less. Every developer should get at least this far.

What can we do about the remaining floating point attributes?

Vertex compression

- Turn floats into halfs?
  - Usually not the best solution
  - Use shorts with scale & bias
    - Unnormalized slightly more accurate (no division by 32767)

```
struct Vertex
{
    short4 Position;    // 8 bytes, 2 unused
    short4 TexCoord;    // 8 bytes
    ubyte4 Tangent;     // 4 bytes
    ubyte4 Bitangent;   // 4 bytes
    ubyte4 Normal;      // 4 bytes
    ubyte4 Color;       // 4 bytes
};          // Total:    32 bytes, 6 attributes
```

An initial idea is to use halfs, but that is almost never the best solution. The logarithmic distribution of values is not a property that is meaningful to either vertex positions or texture coordinates. We want uniform distribution, so shorts is the most sensible choice. We usually need a range outside of the [-1, 1], so we apply a scale and bias. This is derived from the bounding box of the model.

Note that we introduce some small amount of quantization errors when we do this packing. This is usually not much of a problem per se, but when there are different models or parts that stick together but have their own bounding boxes, then this can introduce a slight overlap (occasionally noticeable due to an amount of z-fighting) or a small gaps (very visible). This was solved by sharing settings between models and parts, essentially by picking a bounding box fitting all models that belong together. This actually increases quantization errors, but makes it consistent and thus eliminates the problem.

After applying this optimization we are down to 32bytes, of which two are up for grabs.

Tangent-space compression

SIGGRAPH2012

- Just Cause 2 – RG32F – 8bytes

```
float3 tangent = frac(    In.Tangents.x  * float3(1,256,65536)) * 2 - 1;
float3 normal  = frac(abs(In.Tangents.y) * float3(1,256,65536)) * 2 - 1;
float3 bitangent = cross(tangent, normal);
bitangent = (In.Tangents.y > 0.0f)? bitangent : -bitangent;
```

```
struct Vertex
{
    short4 Position;    // 8 bytes, 2 unused
    short4 TexCoord;    // 8 bytes
    float2 Tangents;    // 8 bytes
    ubyte4 Color;       // 4 bytes
};          // Total:    28 bytes, 4 attributes
```

In Just Cause 2 we went a step further, packing tangent-space into 8 bytes. The first thing to realize is that we usually don't actually need to store all three tangent vectors, we can just derive the third one from the other two. So that shaves off another attribute and 4 bytes. Further we packed the remaining two vectors into floats. This does not reduce the storage space, but takes away another vertex attribute. Trading a small amount of ALU for one less vertex attribute turned out a performance gain in practice.

After this step we are down to 28 bytes and 4 attributes.

Tangent-space in 4 bytes

- Longitude / latitude
  - R,G ⇒ Tangent
  - B,A ⇒ Bitangent
- Trigonometry heavy
  - Fast in vertex shader

```
float4 angles = In.Tangents * PI2 - PI;
float4 sc0, sc1;
sincos(angles.x, sc0.x, sc0.y);
sincos(angles.y, sc0.z, sc0.w);
sincos(angles.z, sc1.x, sc1.y);
sincos(angles.w, sc1.z, sc1.w);
float3 tangent   = float3(sc0.y * abs(sc0.z), sc0.x * abs(sc0.z), sc0.w);
float3 bitangent = float3(sc1.y * abs(sc1.z), sc1.x * abs(sc1.z), sc1.w);
float3 normal    = cross(tangent, bitangent);
normal = (angles.w > 0.0f)? normal : -normal;
```

After Just Cause 2 we further improved on the tangent-space packing. This is a solution that requires only four bytes for the whole tangent-space. We have been using this technique for a number of shaders since. A single vector is encoded into two components as longitude and latitude. The decoding is quite trigonometry heavy, but this is actually not much of a problem in a vertex shader, certainly not as bad as this would look like in CPU code. Not only is trigonometry fast, but on some hardware it actually runs in a separate lane and can be co-issued with other instructions, often more or less hiding the cost entirely in real shaders. This shader actually runs in about the same performance as the previous solution, but at four bytes less.

Tangent-space in 4 bytes

- Quaternion
  - Orthonormal basis

```
void UnpackQuat(float4 q, out float3 t, out float3 b, out float3 n)
{
  t = float3(1,0,0) + float3(-2,2,2)*q.y*q.yxw + float3(-2,-2,2)*q.z*q.zwx;
  b = float3(0,1,0) + float3(2,-2,2)*q.z*q.wzy + float3(2,-2,-2)*q.x*q.yxw;
  n = float3(0,0,1) + float3(2,2,-2)*q.x*q.zwx + float3(-2,2,-2)*q.y*q.wzy;
}

float4 quat = In.TangentSpace * 2.0f - 1.0f;
UnpackQuat(rotated_quat, tangent, bitangent, normal);
```

Another solution we have started to use in a few places is tangent-space represented as a quaternion. This is also just four bytes. The decoding is very straight GPU-friendly vector math. However, the first iteration of this actually ended up being a couple of instructions longer than the previous method.

Tangent-space in 4 bytes

- Rotate quaternion instead of vectors!

```
// Decode tangent-vectors
...
// Rotate decoded tangent-vectors
Out.Tangent   = mul(tangent,   (float3x3) World);
Out.Bitangent = mul(bitangent, (float3x3) World);
Out.Normal    = mul(normal,    (float3x3) World);
```

```
// Rotate quaternion, decode final tangent-vectors
float4 quat = In.TangentSpace * 2.0f - 1.0f;
float4 rotated_quat = MulQuat(quat, WorldQuat);
UnpackQuat(rotated_quat, Out.Tangent, Out.Bitangent, Out.Normal);
```

The first thing we typically do after decoding the vectors is to rotate them with the rotation part of the world matrix. This is another 9 vector instructions, so fairly significant in this context. What we realized is that with a quaternion we can actually rotate the quaternion itself, we just need to pass the world rotation as another quaternion, and then it is just a quaternion multiplication. We can then simply decode it straight to the outputs. This way this technique turned out significantly shorter than the previous method.

The downside of quaternions is that you can only have an orthonormal basis, i.e. no skew is possible. This may not matter much in practice for lighting purposes. Certainly for us it has so far not really matters, but your mileage may vary.

Tangent-space compression

- ”Final” vertex

```
struct Vertex
{
    short4 Position;    // 8 bytes, 2 unused
    short4 TexCoord;    // 8 bytes
    ubyte4 Tangents;    // 4 bytes
    ubyte4 Color;       // 4 bytes
};          // Total:    24 bytes, 4 attributes
```

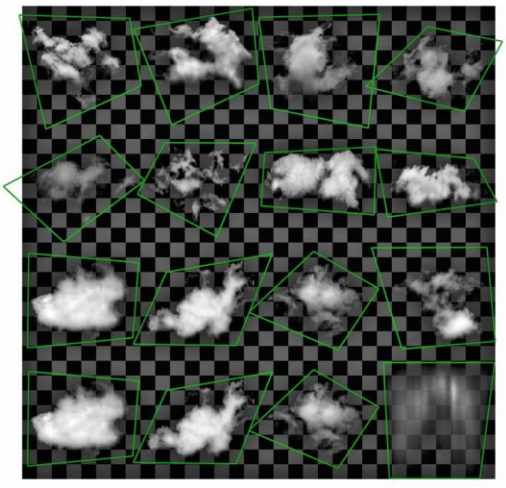
- Other possibilities
  - R5G6B5 color in Position.w
  - Position as R11G11B10

Our ”final” vertex here is then 24 bytes with 4 attributes, of which two are available for use. We have at times tinkered with a few other ideas, but they have never really stuck, but could be worth exploring for some set of models. A 16bit color may be enough and could be stored in Position.w, saving another 4 bytes and one attribute. 10 bit positions could be enough for a subset of models, say for instance characters, if they are always human sized.

Particle trimming

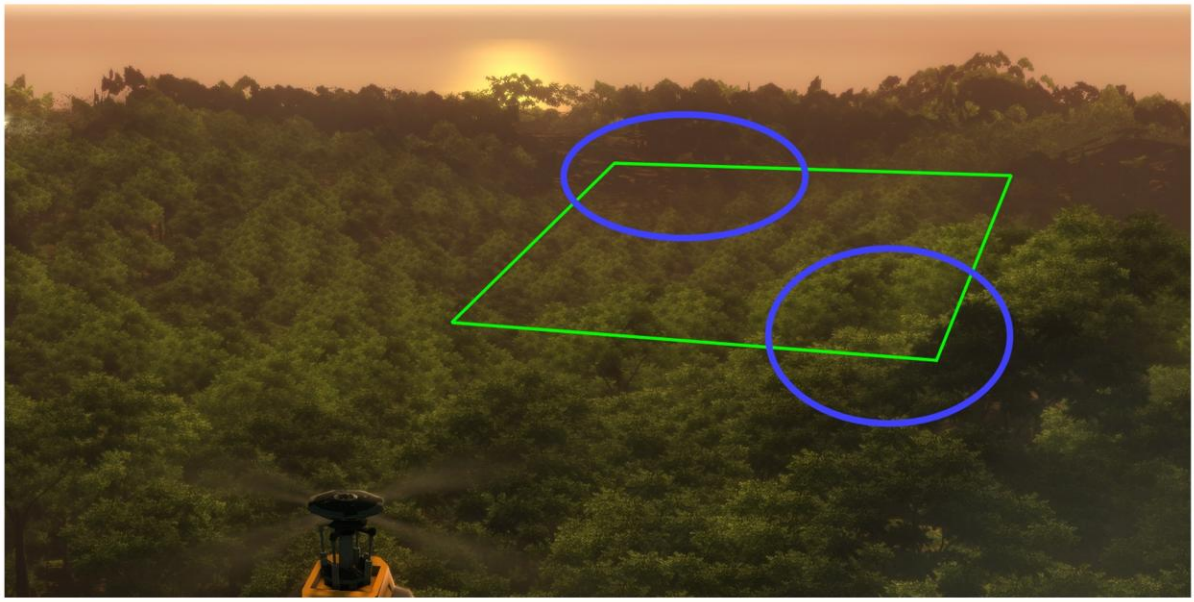
SIGGRAPH2012

- Plenty of alpha = 0 area
  - Find optimal enclosing polygon
  - Achieved > 2x performance!
- *Advances in Real-Time Rendering in Games*
  - ***Graphics Gems for Games: Findings From Avalanche Studios***
  - **515AB, Wednesday 4:20 pm**
- *"Particle Trimmer."* Persson, 2009. [4]



We come up with a particle trimming algorithm for reducing the amount of fillrate waste when rendering particles. This method ended up improving our cloud and particle rendering performance by more than 2x. For details, refer to the ”Graphics Gems for Games: Findings From Avalanche Studios” talk in the “Advances in Real-Time Rendering In Games” course.

Vertex shader culling



Sometimes you batch a lot of simple geometry together into a single draw-call, but want to cull individual instances. An example is this screenshot. Assuming that green rectangle represents a patch of billboard trees, you may have a hundred trees in there, so culling each tree individually or having separate draw calls for each is not optimal. But some of the instances will be completely faded away, for instance in the far end it could be fading over to the coarse distant forest representation and close up it could be faded over to an actual fully detailed model, so you want to avoid drawing those billboards that are completely faded away.

Vertex shader culling

- Sometimes want to cull at vertex level
  - Especially useful within a single draw-call
    - Foliage, particles, clouds, light sprites, etc.
  - Example: 100 foliage billboards, many completely faded out…
  - Throw things behind far plane!

```
Out.Position = ...;
float alpha = ...;

if (alpha <= 0.0f)
    Out.Position.z = -Out.Position.w; // z/w = -1, behind far plane
```

We used a simple but effective vertex level culling approach for many of our systems, including foliage, clouds, the distant light system etc. The basic idea is to just throw things behind the far plane such that it will be culled that way. This can be done by simply assigning –w to z, such that z/w = -1, which is behind the far plane with a reversed depth buffer. With a regular depth buffer you would use for instance 2*w, so that z/w = 2.

Early on we made the mistake to write a completely degenerate vertex to the output, such as for instance float4(0, 0, 0, 0) or float4(-1, -1, -1, 1). This works fine if all three vertices within a triangle gets culled. However, small variations between vertices may occasionally cause for instance two vertices to survive but the third one being culling. If you have two valid and one destroyed vertex you may instead get a huge degenerate triangle covering half the screen. If you are lucky you will notice artifacts from this. If you are unlucky, it may be completely faded away and not be visible at all, only eating up half a screen worth of fillrate resulting in a huge performance penalty instead of an optimization. The point of modifying only z in particular is that the other three components are core parts of the rasterization process. The position on the screen is x/w and y/w, and w is also used for perspective correction so your interpolators and thus texture lookups will also be affected if w is messed with, but z is only used for depth testing. A degenerate z will not change a triangle's position on the screen and thus not affect performance, nor will it mess with perspective correction, so a partly culled triangle should still generally look fine. At worst it will clip through the world geometry back to the far plane, but you normally shouldn't be seeing any of that in practice because your likely more or less entirely faded away anyway.

Draw calls

SIGGRAPH2012

- *"Batch, batch, batch"*. Wloka, 2003. [2]
  - 300 draw calls / frame
- What's reasonable these days?
  - 2-10x faster / thread
  - We're achieving 16,000 @ 60Hz, i7-2600K, single draw thread
- Not so much DrawIndexed() per se
  - Culling
  - Sorting
  - Updating transforms

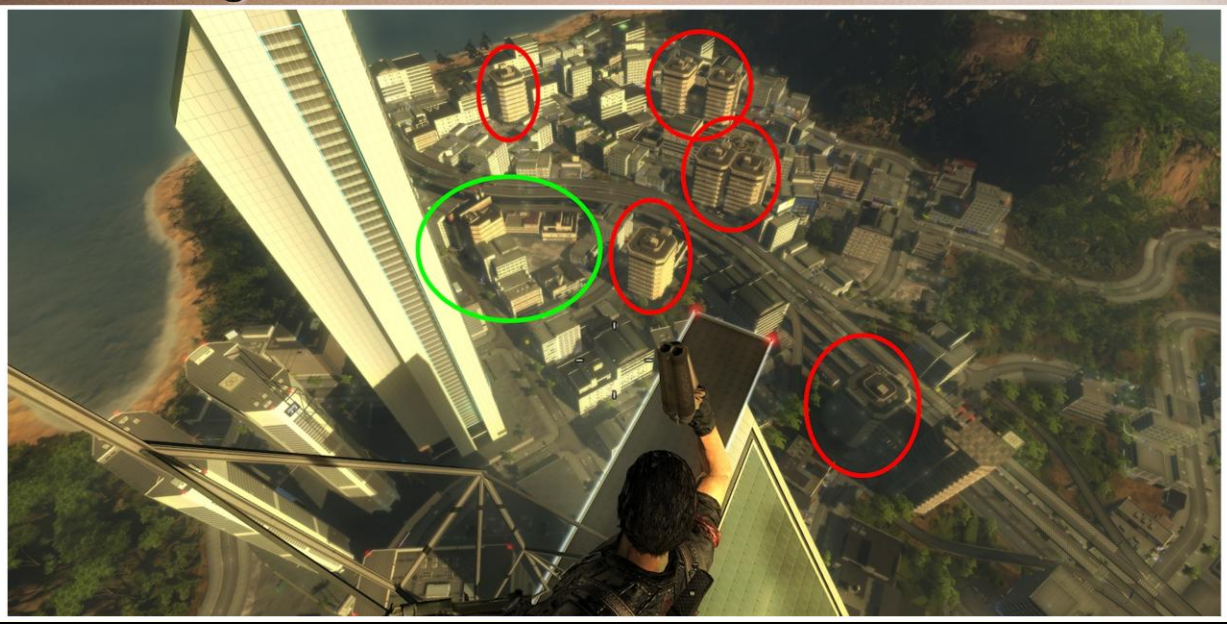
```
ID3D11DeviceContext_DrawIndexed:
mov     eax, dword ptr [esp+4]
lea     ecx, [eax+639Ch]
mov     eax, dword ptr [eax+1E8h]
mov     dword ptr [esp+4], ecx
jmp     eax
```

It is generally my impression that the old "draw calls are enormously expensive - especially on PC" mentality is still widespread in this industry. This used to be true back in the DX9 era, for instance back in 2003 when the "Batch, batch, batch" presentation was given. 300 draw calls / frame was a common recommendation from both major IHVs at this point. I was at ATI in 2004, and can testify that the findings were very much true at that time. This was a time where you could do some profiling on a game build and see 10-20% of the CPU time within D3D and 30-50% within the driver. The overhead was huge.

Fast forward to 2012. CPUs are much faster, even though the single-threaded gain has been sluggish at best the last 5 years or so. But we have also replaced the DX9 API with a much more efficient design in DX10/11, plus a much improved driver model. These days the D3D runtime is essentially just forwarding your draw-calls directly into the user mode driver. The whole implementation is those 5 x86 instructions there. It is even jumping into the user mode driver, not making a call, such that it will return directly to your code from the user mode driver and not even return to D3D at all. With DX11 you also have the opportunity to do multithreading. On my project we haven't actually gone down that path yet, but it notable that even with a single draw thread we can easily render 16,000 draw calls per frame and hit 60fps. We don't actually target that much, but have at different times had that sort of content in our build. On a modern CPU this has not cause any real problems.

Having plenty of instances can still be a problem though. The main overhead normally isn't DrawIndexed() per se, but all of sorts of work typically associated with render instances, for instance culling, sorting, and updating transforms and other per-instance data. The main goal should primarily be reducing instance count, rather than calls to a particular API function.

Reducing draw calls



Even so, there are still cases where a reduction of draw call make sense. We had a large number of draw-calls in the city in Just Cause 2, and this turned problematic on the Xbox360 in particular. The PS3 did not have the same overhead, and the PC just has the raw power to crunch through it anyway.

In this screenshot, you may think that those building in the red circles are prime candidates for instancing, but if you put them together in a single bounding box it will be huge resulting in highly ineffective culling. If you cull each individually you incur a much higher cost for culling plus the cost of dynamically building an instance buffer. What you ideally want to do is merge for instance those buildings in the green circle, but the problem then is that they are completely different meshes.

Reducing draw calls

- Merge-Instancing
  - Multiple meshes and multiple transforms in one draw-call
  - Shader-based vertex data traversal
- *Advances in Real-Time Rendering in Games*
  - ***Graphics Gems for Games: Findings From Avalanche Studios***
  - **515AB, Wednesday 4:20 pm**

So we came up with a method for drawing a bunch of different models with different transforms in a single draw call without duplicating vertex data. For details, refer to the ”Graphics Gems for Games: Findings From Avalanche Studios” talk in the “Advances in Real-Time Rendering In Games” course.

State sorting

- 64-bit sort-id
  - *"Order your graphics draw calls around!"* Ericson, 2008. [3]
  - Render-passes prearranged
    - E.g. ModelsOpaque, ModelsTransparent, PostEffects, GUI etc.
  - Material types prearranged
    - E.g. Terrain, Character, Particles, Clouds, Foliage etc.
- Dynamic bit layout
  - Typically encodes state

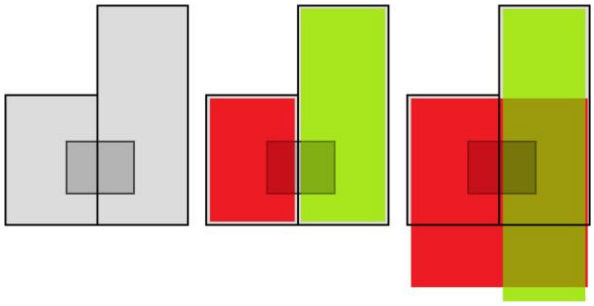
Our state sorting is loosely based on Ericson's approach [3]. We do a few things differently. Our render-passes are always in the same order, so they are prearranged, thus we don't waste any bits on storing that information. We also have our material types pre-arranged. There are several advantages to this. Firstly, we don't waste any bits on this information, leaving more bits for useful state encoding. Secondly, having separate draw lists for per render-pass and material type results in smaller lists to sort, which reduces sorting overhead. It also reduces draw-list contention when adding things to render. But the main advantage is that we get better use of the available bits. We don't have any global layout, but instead it is up to each material type to decide the bit layout of its sort id and interpret the data in it. Different systems have different concerns and typically want to sort on different parameters. With a completely dynamic bit layout each system can make optimal use of the available bits. For instance, terrain may want to sort on depth primarily to optimize for HiZ, whereas characters rarely are much of an occluder nor would it for that matter gain much from HiZ considering that they typically are vertex bound, so it would be a waste of time to even compute their depth values. So instead for characters we want to sort on things that matter there, like vertex declaration and shader LODs.

A nice thing about encoding state into the sort-id is that a lot of the data we need is already included in the sort-id for our instances. So while processing the draw-list we can often issue draw-calls without even looking at much of the data belonging to individual instance, but already know for instance which shader to use from the bits in the sort-id. This makes this approach very cache-friendly.

Culling

SIGGRAPH2012

- BFBC – Brute Force Box Culling
  - Artist placed occluder boxes
    - Culled individually, not union
    - Educate content creators
  - SIMD optimized
  - PPU & SPU work in tandem



- *"Culling the Battlefield."* Collin, 2011. [3]

Our culling system is called BFBC – Brute Force Box Culling. The reason we use a ”brute-force” approach, which in this context means a linear array of instances to cull, is that it is a simple system which is very cache-friendly and can be highly SIMD optimized. Many culling approaches tend to be highly hierarchical. Theoretically they should be faster if you just look at the asymptotic behavior, but in practice they tend to be very branch-heavy and cache-unfriendly, eating up any of the theoretical benefits for realistic workloads. The hierarchies can also be complex to maintain and keep up-to-date, especially if you have a very dynamic world. We can create multiple instances of this system though, and different parts of the engine use their own instances. It is also possible to build coarse level hierarchies on top of this system, should that make sense somewhere.

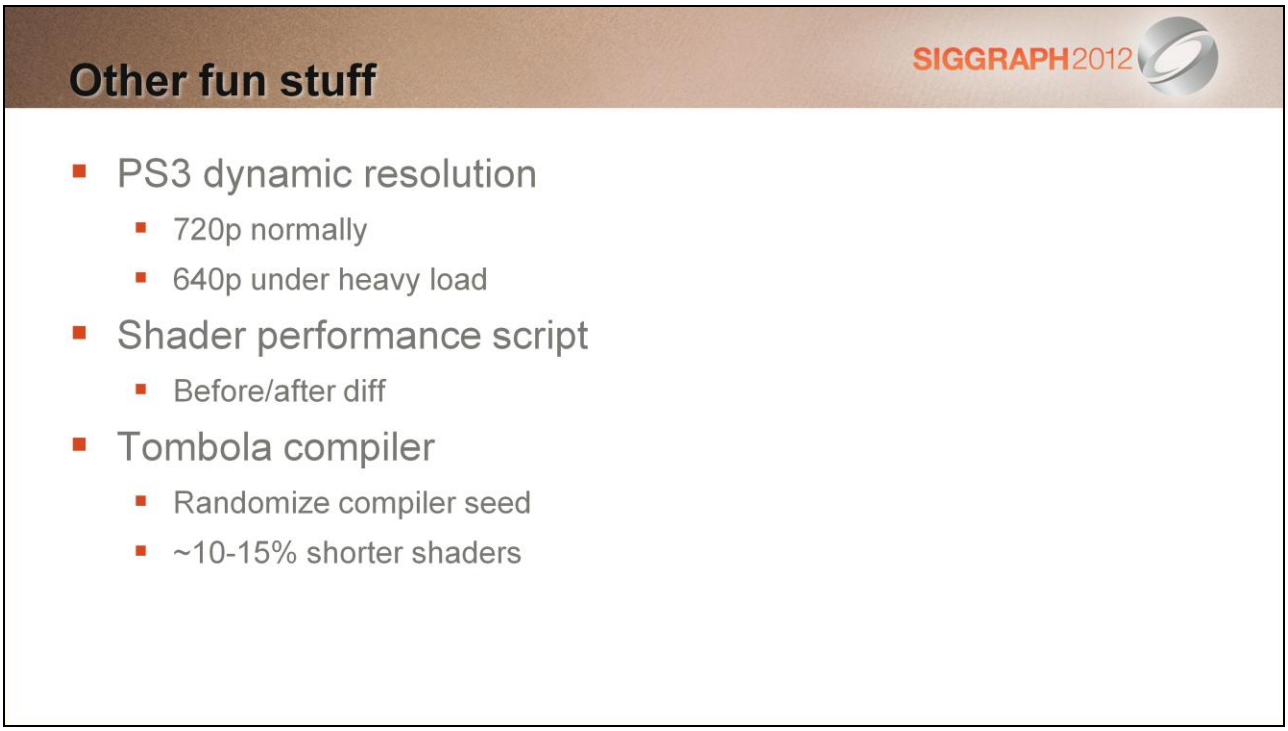
The system is driven by artist placed occluder boxes, typically placed at strategic places, such as within large buildings, hills and other good occluders. We only cull on occluders on a box-by-box case, not by the union of occluders, since the union case is a lot more complicated and would have relatively small gain in practice for our game. Often you can get near the same culling just by educating content creators to create the best possible occluder boxes. The optimal occluder boxes may be unintuitive at first. For instance, consider the two adjacent buildings in the illustration, with an occludee behind it. The initial approach by an inexperienced content creator tends to be something like that in the middle, with two boxes that together fills the entire volume, however, no box does by itself entirely cover the occludee in this case, thus we would actually end up drawing it. So what content creators need to do is on the right, with the red box extending all the way through the green box. There is also no particular reason for the box to stop at the bottom of the building either. It is often very beneficial to extend it deep into the ground, allowing for culling of for instance things that may be at a lower elevation at some distance behind the building, for instance very commonly water patches.

Streaming

- Pre-optimized data on disc
  - Gigabyte sized archives
  - Related resources placed adjacently
  - Zlib compression
- Request ordered by priority first, then adjacency
- Concurrent load and create

Our streaming system uses pre-optimized data on disc, using gigabyte sized archives. To access all the game data we only need to have a handful of handles open to the file system at any time. To reduce the number of seeks we place related data adjacently on disc and also order requests by adjacency for requests of same priority.

Other fun stuff



On the PS3 we implemented a dynamic resolution scaling system. Normally you have your full 720p, but if there is a lot of action and performance drops we continuously scale resolution down to at most 640p to compensate. Normally you don't notice the reduction in resolution. This way we were able to keep performance at 30fps under almost any circumstances.

We normally use über-shaders that are compiled down to a set of optimized shader LODs depending on enabled features etc. Within the visuals team we used a shader performance script to keep track of the impact of shader changes we made. It compiled all shader LODs and compared the result with the previous revision, returning a list of changes, such as instruction count and register use on all platforms. As a rule we always attached this log to our code reviews whenever they included a change to a shader. This way we always kept performance on top of our minds during development and knew the impact the changes had, at least at the shader level. Of course, we typically also included a comment on the actual performance impact in terms of milliseconds in some arbitrary test location as well.

The PS3 fragment shader hardware is rather esoteric in many ways. The compiler has a really hard nut to crack trying to optimally map a general shader onto that instruction set. A consequence of that is that trivial changes sometimes result in wildly different outcome in terms of shader length and/or register use. However, you can feed the compiler a seed that will affect the heuristics it will use and whatever paths it takes internally in trying to arrange things. It is very common that a custom seed exists that will generate a faster shader than the defaults. The problem is just coming up with the optimal one. We made a tombola compiler that essentially randomized seeds and tried them, and whenever it found a better seed than previously encountered it would check that into perforce. So whenever you compiled the shader you got the currently known best seed. We basically had a machine standing there continuously just trying random seeds for all our shader LODs and continuously updating the database with its latest results. After weeks of repeated attempts we would arrive at some sort of optimum with little new improvements coming in (until someone checked in a change to a shader that is, which reset the database for that shader). In the end we typically got shaders that were 10-15% shorter than with default settings, resulting in a pretty decent performance

References

[1] http://en.wikipedia.org/wiki/Gauss%E2%80%93Jordan_elimination
[2] http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf
[3] http://realtimecollisiondetection.net/blog/?p=86
[4] http://www.humus.name/index.php?page=Cool&ID=8
[5] http://publications.dice.se/attachments/CullingTheBattlefield.pdf
[6] http://www.altdevblogaday.com/2012/01/05/tricks-with-the-floating-point-format/
[7] http://fgiesen.wordpress.com/2010/10/21/finish-your-derivations-please/

Thank you!

Emil Persson
Avalanche Studios


@_Humus_

Bonus slides!

Yay! ☺

Random rants / pet peeves

SIGGRAPH2012

- Shader mad-ness (pun intended)
  - Understand the hardware
  - (x + a) * b ⇒ x * b + c, where c = a*b
  - a / (x - b) ⇒ 1.0f / (c * x + d), where c = 1/a, d = -b/a

```
// Constants
C = { f / (f-n), n / (n-f) };

// SUB-RCP-MUL
float GetLinearDepth(float z)
{
    return C.y / (z - C.x);
}
```

```
// Constants
C = { f / n, 1.0f - f / n };

// MAD-RCP
float GetLinearDepth(float z)
{
    return 1.0f / (z * C.y + C.x);
}
```

  - ”Finish your derivation” [7]

It is unfortunate and highly annoying to see that smart and experienced people constantly get this wrong. Everyone ought to know that the hardware has multiply-add (MAD) units and not add-multiply. Thus expressions written in MAD form will compile to shorter and faster code. It is usually trivial to make that transform, which makes it even more annoying when it is not done. And no, the shader compiler will not optimize it for you. It can’t. It’s unsafe and against the rules. You wouldn’t want it do it, because that would potentially break some shaders. Just write your shaders right instead.

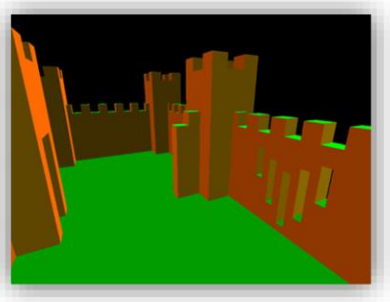
The same applies to many other kinds of math also. Understanding the hardware instruction set allows you make faster code, even though you’re working in a high level language. At Siggraph last year I saw this sort of conversion code for turning a depth value into a linear view distance, at least twice from different people. Smart people, being speakers on Siggraph and all, experienced, knowledgeable and reputable. Yet failed to observe the trivial transformation of the code that would make it compile to a shorter sequence of instructions. This makes me sad.

This is kind of related to Fabian Giesen’s rant on finishing your derivations. He was ranting on half-assed math though, and I agree on every point he made. My rant here is essentially the same, but for shaders.

Random rants / pet peeves

SIGGRAPH2012

- Merge linear operations into matrix
- Understand depth
  - Non-linear in view direction
  - Linear in screen-space!
- Premature generalizations
  - ಠ_ಠ
  - Don't do it!
  - YAGNI



I often see matrix math followed or preceded by other linear operations. A typical example is computing a position through a matrix, then subtracting for instance a light position. Just merge that subtraction into the matrix!

Everyone knows that depth has a non-linear distribution of values in the view direction. What is apparently not universally known is that it is linear in screen-space. At last Siggraph I saw a presentation, again a smart, knowledgeable speaker with clever tricks and solutions, do very odd math with depth that could only come out of an unclear idea about the nature of depth values. Understanding linearity in screen-space could have turned his approach into something much more sensible.

Premature generalizations – Just do things when you need them, not what you think will be needed later. Solve your specific problem, not the general case. We still suffer from a lot of early abstractions done in our code base which have been hard to fix. No, we will never inherit anything from our GraphicsEngine class. It does not need an abstract base class and virtual functions.

Process

- Design for performance
  - ”~~Premature optimizations is the root of all evil~~”

    *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet **we should not pass up our opportunities in that critical 3%**. A good programmer will not be lulled into complacency by such reasoning, he will be wise to **look carefully at the critical code**; but only **after that code has been identified**"*
- Code reviews
- Profile every day

Design for performance. Profile continuously. Stay on top of current performance. The worse your performance is, the more likely it is that you will add even more unreasonable workloads to the engine. If you are running at 10fps and add something ridiculous like 5ms for something minor, you will now run at 9.5fps and nobody will notice the difference. If you were running at 30fps, you'd drop to 26fps. Chances are someone may notice it, if not otherwise then at least on the fps counter. If you were running at 60fps, you'd drop to 46fps. Everyone would start yelling. Always keep yourself at a range where new lousy code gets detected early.

Whenever someone's misquoting Knuth, point them the full context and then educate them on the consequences of letting performance slide.

Always do code reviews for anything that's not absolutely trivial. Nothing will do more for improving code quality. Hacks may still occasionally be necessary, but at least you will have to motivate them to your team members and there will be more than one person aware of their existence, which increases the likelihood that it will eventually be replaced by proper code.