

Far Cry and DirectX

Carsten Wenzel



Far Cry uses the latest DX9 features

- **Shader Models 2.x / 3.0** ✓
 - Except for vertex textures and dynamic flow control ✗
- **Geometry Instancing** ✓
- **Floating-point render targets** ✓



Dynamic flow control in PS

- To consolidate multiple lights into one pass, we ideally would want to do something like this...

```
float3 finalCol = 0;
float3 diffuseCol = tex2D( diffuseMap, IN.diffuseUV.xy );
float3 normal = mul( IN.tangentToWorldSpace,
                    tex2D( normalMap, IN.bumpUV.xy ).xyz );
for( int i = 0; i < cNumLights; i++ )
{
    float3 lightCol = LightColor[ i ];
    float3 lightVec = normalize( cLightPos[ i ].xyz - IN.pos.xyz );
    // ...
    // Attenuation, Specular, etc. calculated via if( const_boolean )
    // ...
    float nDotL = saturate( dot( lightVec.xyz, normal ) );
    final += lightCol.xyz * diffuseCol.xyz * nDotL * atten;
}
return( float4( finalCol, 1 ) );
```



Dynamic flow control in PS

- **Welcome to the real world...**
 - **Dynamic indexing only allowed on input registers; prevents passing light data via constant registers and index them in a loop**
 - **Passing light info via input registers not feasible as there are not enough of them (only 10)**
 - **Dynamic branching is not free**



Loop unrolling

- We chose not to use dynamic branching and loops
- Used static branching and unrolled loops instead
- Works well with Far Cry's existing shader framework
- Shaders are precompiled for different light masks
 - 0-4 dynamic light sources per pass
 - 3 different light types (spot, omni, directional)
 - 2 modification types per light (specular only, occlusion map)
- Can result in over 160 instructions after loop unrolling when using 4 lights
 - Too long for ps_2_0
 - Just fine for ps_2_a, ps_2_b and ps_3_0!
- To avoid run time stalls, use a pre-warmed shader cache



How the shader cache works

- **Specific shader depends on:**
 - 1) **Material type**
(e.g. skin, phong, metal)
 - 2) **Material usage flags**
(e.g. bump-mapped, specular)
 - 3) **Specific environment**
(e.g. light mask, fog)



How the shader cache works

- **Cache access:**
 - Object to render already has shader handles? Use those!
 - Otherwise try to find the shader in memory
 - If that fails load from harddisk
 - If that fails generate VS/PS, store backup on harddisk
 - Finally, save shader handles in object
- **Not the ideal solution but**
 - Works reasonably well on existing hardware
 - Was easy to integrate without changing assets
- **For the cache to be efficient...**
 - All used combinations of a shader should exist as pre-cached files on HD
 - On the fly update causes stalls due to time required for shader compilation!
 - However, maintaining the cache can become cumbersome



Loop unrolling - Pros/Cons

- **Pros:**
 - **Speed! Not branching dynamically saves quite a few cycles**
 - **At the time, we found switching shaders to be more efficient than dynamic branching**
- **Cons:**
 - **Needs sophisticated shader caching, due to number of shader combinations per light mask (244 after presorting of combinations)**
 - **Shader pre-compilation takes time**
 - **Shader cache for Far Cry 1.3 requires about 430 MB (compressed down to ~23 MB in patch exe)**



Geometry Instancing

- Potentially saves cost of $n-1$ draw calls when rendering n instances of an object
- Far Cry uses it mainly to speed up vegetation rendering
- Per instance attributes:
 - Position
 - Size
 - Bending info
 - Rotation (only if needed)
- Reduce the number of instance attributes! Two methods:
 - Vertex shader constants
 - Use for objects having more than 100 polygons
 - Attribute streams
 - Use for smaller objects (sprites, impostors)



Instance Attributes in VS Constants

- Best for objects with large numbers of polygons, prevents GPU from becoming attribute bound (see Cem's talk)
- Put instance data in VS constants and index into additional stream
 - WGF 2.0 will support an automatically generated instance index!
- Large batches need to be split up to fit attributes in VS constant (try to fit attributes for at least eight instances to amortize startup cost!)
- Use *SetStreamSourceFrequency* to setup geometry instancing as follows...

```
SetStreamSourceFrequency( geomStream,  
    D3DSTREAMSOURCE_INDEXEDDATA | numInstances );  
SetStreamSourceFrequency( instStream,  
    D3DSTREAMSOURCE_INSTANCEDATA | 1 );
```

- Be sure to reset the vertex stream frequency once you're done, *SSSF(strNum, 1)*!



VS Snippet to unpack attributes (position & size) from VS constants to create matMVP and transform vertex

```
const float4x4 cMatViewProj;  
const float4 cPackedInstanceData[ numInstances ];  
  
float4x4 matWorld;  
float4x4 matMVP;  
  
int i = IN.InstanceIndex;  
  
matWorld[ 0 ] = float4( cPackedInstanceData[ i ].w, 0, 0, cPackedInstanceData[ i ].x );  
matWorld[ 1 ] = float4( 0, cPackedInstanceData[ i ].w, 0, cPackedInstanceData[ i ].y );  
matWorld[ 2 ] = float4( 0, 0, cPackedInstanceData[ i ].w, cPackedInstanceData[ i ].z );  
matWorld[ 3 ] = float4( 0, 0, 0, 1 );  
  
matMVP = mul( cMatViewProj, matWorld );  
OUT.HPosition = mul( matMVP, IN.Position );
```



Instance Attribute Streams

- Original geometry instancing approach... “Only pay the cost for 1 draw call when rendering n instances”
- Best for objects with few polygons, less likely to become attribute bound
- Put per instance data into additional stream
- Setup vertex stream frequency as before and reset when you’re done



VS Snippet to unpack attributes (position & size) from attribute stream to create matMVP and transform vertex

```
const float4x4 cMatViewProj;  
  
float4x4 matWorld;  
float4x4 matMVP;  
  
matWorld[ 0 ] = float4( IN.PackedInstData.w, 0, 0, IN.PackedInstData.x );  
matWorld[ 1 ] = float4( 0, IN.PackedInstData.w, 0, IN.PackedInstData.y );  
matWorld[ 2 ] = float4( 0, 0, IN.PackedInstData.w, IN.PackedInstData.z );  
matWorld[ 3 ] = float4( 0, 0, 0, 1 );  
  
matMVP = mul( cMatViewProj, matWorld );  
OUT.HPosition = mul( matMVP, IN.Position );
```



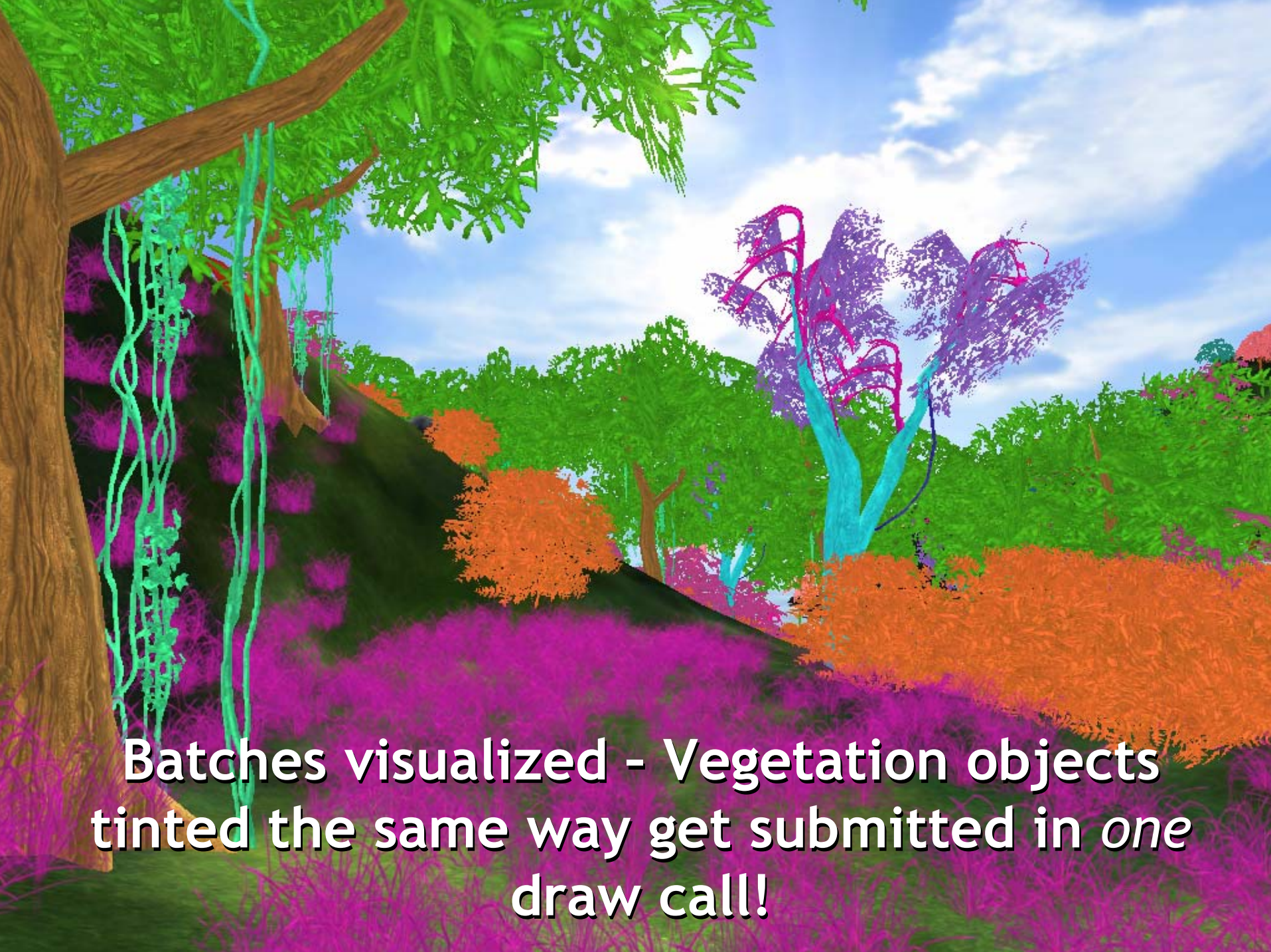
Geometry Instancing - Results

- Depending on the amount of vegetation, rendering speed increases up to 40% (when heavily draw call limited)
- Allows us to increase sprite distance ratio, a nice visual improvement with only a moderate rendering speed hit





Scene drawn normally



Batches visualized - Vegetation objects tinted the same way get submitted in *one* draw call!

000 0

High Dynamic Range Rendering



High Dynamic Range Rendering

- Uses A16B16G16R16F render target format
- Alpha blending and filtering is essential
- Unified solution for post-processing
 - Glare, flares, etc. can be added more naturally



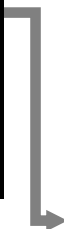
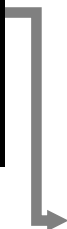
HDR - Implementation

- HDR in Far Cry follows standard approaches
 - Kawase's bloom filters
 - Reinhard's tone mapping operator
 - See DXSDK sample
- Performance hint
 - For post processing try splitting your color into rg, ba and write them into two MRTs of format G16R16F. That's more cache efficient on some cards.

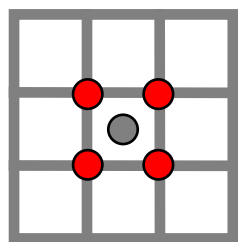


Bloom from [Kawase03]

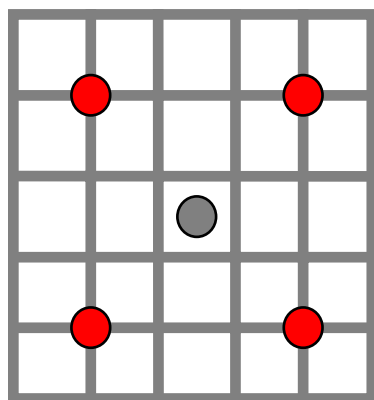
- Repeatedly apply small blur filters
- Composite bloom with original image
 - Ideally in HDR space, followed by tone mapping



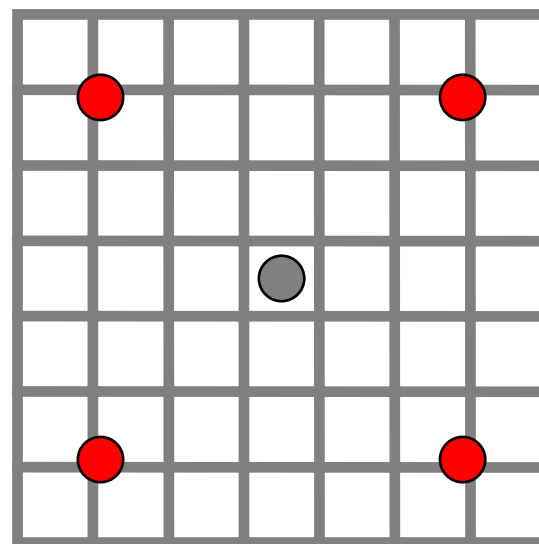
Increase Filter Size Each Pass



1st pass



2nd pass



3rd pass

- Pixel being Rendered
- Texture sampling points

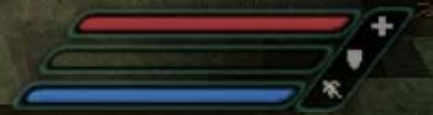
From [Kawase03]



GOD 0

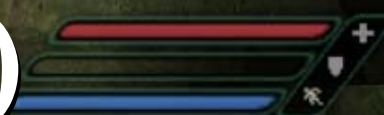


No HDR



GOD 0

HDR (tone mapped scene +
bloom + stars)



[Reinhard02] - Tone Mapping

1) Calculate scene luminance

On GPU done by sampling the log() values, scaling them down to 1x1 and calculating the exp()

$$Lum_{avg} = \exp \left(\frac{1}{N} \sum_{x,y} \log(\delta + Lum(x,y)) \right)$$

2) Scale to target average luminance α

$$Lum_{scaled}(x,y) = \frac{\alpha}{Lum_{avg}} Lum(x,y)$$

3) Apply tone mapping operator

$$Color(x,y) = \frac{Lum_{scaled}(x,y)}{1 + Lum_{scaled}(x,y)}$$

- To simulate light adaptation replace Lum_{avg} in step 2 and 3 by an adapted luminance value which slowly converges towards Lum_{avg}
- For further information attend Reinhard's session called "Tone Reproduction In Interactive Applications" this Friday, March 11 at 10:30am



HDR - Watch out

- Currently no FSAA
 - Extremely fill rate hungry
 - Needs support for float buffer blending
 - HDR-aware production¹ :
 - Light maps
 - Skybox
- 1) For prototyping, we actually modified our light map generator to generate HDR maps and tried HDR skyboxes. They look great. However we didn't include them in the patch because...
- Compressing HDR light map textures is challenging
 - Bandwidth requirements would have been even bigger
 - Far Cry patch size would have been huge
 - No time to adjust and test all levels



Conclusion

- **Dynamic flow control in ps_3_0**
- **Geometry Instancing**
- **High Dynamic Range Rendering**



References

- [\[Kawase03\]](#) Masaki Kawase, “Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless),” Game Developer’s Conference 2003
- [\[Reinhard02\]](#) Erik Reinhard, Michael Stark, Peter Shirley and James Ferwerda, “Photographic Tone Reproduction for Digital Images,” SIGGRAPH 2002.



Questions?

carsten@crytek.de

Thanks to...

Martin Mittring & Andrey Khonich @ Crytek

Jason Mitchell & Richard Huddy @ ATI

