Supplementary Document for Moment Transparency

Brian Sharpe
Weta Digital
Wellington, New Zealand
bsharpe@wetafx.co.nz

ACM Reference Format:
Brian Sharpe. 2018. Supplementary Document for Moment Transparency. In *HPG '18: High-Performance Graphics, August 10–12, 2018, Vancouver, Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3231578.3231585

1 IMPLEMENTATION DETAILS

We now give implementation details for moment transparency (MT). First we present code for the standard MT algorithm (Section 1.1), followed by code for two moments passes (Section 1.2), and finish with code for using non-linear quantized moments (NLQM) [Peters 2017] with *overestimation* [Peters et al. 2017] (Section 1.3).

Because MT extends weighted blended order-independent transparency (WBOIT), and moment shadow mapping (MSM), we will only document the areas of MT which are different (i.e. Stages 2 and 3). For a more thorough description of how WBOIT or MSM are implemented, we refer the reader to [McGuire and Bavoil 2013] and [Peters and Klein 2015; Peters et al. 2017] respectively.

1.1 Implementing Moment Transparency

To handle Stage 2 of MT, rasterize all transparent geometry, calling *WriteMoments(...)* for each fragment.

```
// precompute C0 = 1.0 / nearPlane
// precompute C1 = 1.0 / log( farPlane / nearPlane )
float DepthToUnit( float z, float C0, float C1 )
{
  return log( z * C0 ) * C1;
}
vec4 MakeMoments4( float z )
{
  float zsq = z * z;
  return vec4( z, zsq, zsq * z, zsq * zsq );
}
void WriteMoments(
  float z,
  float alpha,
  out vec4 o_moments,        // write to FP32_RGBA as additive
  out float o_opticalDepth ) // write to FP32_R as additive
{
  const float kMaxAlpha = 1.0 - 0.5/256.0; // clamp alpha
  float opticalDepth = -log( 1.0 - ( alpha * kMaxAlpha ) );
  float unitPos = DepthToUnit( z, C0, C1 );
  o_moments = MakeMoments4( unitPos ) * opticalDepth;
  o_opticalDepth = opticalDepth;
}
```

As can be seen, alpha is clamped to a value below 1.0 for two reasons: log(0) is undefined; and alpha values close to 1.0 produce large optical depth, which can cause precision issues when accumulating many fragments. Our experiments have shown $1.0 - 0.5/256$ to be reasonable.

For Stage 3 we need to replace the standard WBOIT weight function $w(z, \alpha)$ with our new moments-based version:

```
float Hamburger4MSM( vec4 moments, float z )
{
  //  EstimateIntegralFrom4Moments and
  //  Compute4MomentUnboundedShadowIntensity implementation, and
  //  associated bias can be found in the demo code at
  //  http://jcgt.org/published/0006/01/03/
  //  use Compute4MomentUnboundedShadowIntensity for basic MT
  //  use EstimateIntegralFrom4Moments for MT with overestimation
  moments = mix(moments,vec4(0.0f,0.375f,0.0f,0.375f),3.0e-7f);
  float result;
  Compute4MomentUnboundedShadowIntensity(result, moments, z, 0.0);
  return result;
}
float w( float z, float alpha )
{
  vec4 moments = ...;  // read from FP32_RGBA moment texture
  float totalOD = ...; // read from FP32_R opticalDepth texture
  float unitPos = DepthToUnit( z, C0, C1 );

  if ( totalOD != 0.0 )
    moments /= totalOD;  // normalize
  float ma = Hamburger4MSM( moments, unitPos );
  ma = exp( -ma * totalOD );

  return ma * alpha;
}
```

1.2 Implementing Two Moments Passes

To handle two moments passes, we need to track two sets of moments simultaneously. For this we allocate four buffers: two FP32_RGBA buffers for the moments and two FP32_R buffers for the optical depth totals. As with standard MT, all buffers should be initialized to zero and blend mode set to additive.

After Stage 2, we require an additional rasterization stage, which builds a new set of moments, while simultaneously sampling those generated by Stage 2. The code for this is as follows:

```
void WriteSecondMoments(
  float z,
  float alpha,
  out vec4 o_moments,        // write to 2nd FP32_RGBA as additive
  out float o_opticalDepth ) // write to 2nd FP32_R as additive
{
  const float kMaxAlpha = 1.0 - 0.5/256.0; // clamp alpha
  float opticalDepth = -log( 1.0 - ( alpha * kMaxAlpha ) );
  float unitPos = DepthToUnit( z, C0, C1 );

  // occlude optical depth by first set of moments
  vec4 moments = ...;  // read 1st FP32_RGBA moment texture
  float totalOD = ...; // read 1st FP32_R opticalDepth texture
  if ( totalOD != 0.0 )
     moments /= totalOD;  // normalize
  opticalDepth *= exp( -Hamburger4MSM(moments,unitPos) * totalOD);
```


HPG '18, August 10–12, 2018, Vancouver, Canada

ACM ISBN 978-1-4503-5896-5/18/08…$15.00
https://doi.org/10.1145/3231578.3231585

```
  o_moments = MakeMoments4( unitPos ) * opticalDepth;
  o_opticalDepth = opticalDepth;
}
```

Because the second set of moments has been occluded by the first, it will have an incorrect total accumulated alpha. To fix this we rescale the transmittance sample by the ratio of the two total accumulated alphas. This requires a different $w(z, \alpha)$ implementation for Stage 3:

```
float w( float z, float alpha )
{
  float totalOD1 = ...; // read 1st FP32_R opticalDepth texture
  float totalOD2 = ...; // read 2nd FP32_R opticalDepth texture
  vec4 moments = ...;   // read 2nd FP32_RGBA moment texture
  float unitPos = DepthToUnit( z, C0, C1 );

  if ( totalOD2 != 0.0 )
    moments /= totalOD2;  // normalize
  float ma = Hamburger4MSM( moments, unitPos );
  ma = exp( -ma * totalOD2 );

  // alpha from the second moments needs rescaled to match
  // the first total accumulated alpha
  ma = 1.0 - ma;
  ma *= ( 1.0 - exp( -totalOD1 ) ) /
        max( 1.0 - exp( -totalOD2 ), 1e-4 );
  ma = 1.0 - ma;

  return ma * alpha;
}
```

1.3 Implementing Non-Linear Quantized Moments With Overestimation

We now show how non-linear quantized moments (NLQM) can be used with overestimation. This follows directly from the works of [Peters 2017], and we direct the reader there for any further information.

```
//
//  This follows directly from Non-Linear Quantized Moments
//  We direct the reader there for any further details
//  http://cg.cs.uni-bonn.de/en/publications/paper-details/peters-2017-quantized-msm/
//  http://cg.cs.uni-bonn.de/aigaion2root/attachments/NonLinearMSMCode.zip
//

float MomentMad( float x, float y, float w ) { return x * y + w; }

//  utility for Compute4MomentNonLinearOverEstimation
float Compute4MomentNonLinearRootWeight(
    float Root,
    vec3 InvVals )
{
    float FlippedRoot=1.0f-Root;
    float RootSquared=Root*Root;
    float FlippedRootSquared=FlippedRoot*FlippedRoot;
    float RootWeight=1.0f/dot(
        InvVals,
        vec3(FlippedRootSquared,RootSquared,RootSquared*FlippedRootSquared));
    return RootWeight;
}

//  analog of Compute4MomentNonLinearLowerBound, but allows for overestimation
float Compute4MomentNonLinearOverEstimation(
    float IntervalEnd,
    vec2 Support,
    float Weight,
    float FourthMomentOffset,
    float OverEstimationWeight)     //  0.0 == lower bound  1.0 == upper bound
{
    // Avoid zero variance
    const float ClampingOffset=1.0e-6f;
    Weight=clamp(Weight,ClampingOffset,1.0f-ClampingOffset);
    Support.y=max(Support.x+ClampingOffset,Support.y);

    // Normalize the interval end such that Support can be treated as
    // vec2(0.0f,1.0f). The fourth moment offset needs to be scaled accordingly.
    float Scaling=1.0 / (Support[1]-Support[0]);
    float NormalizedIntervalEnd=(IntervalEnd-Support[0])*Scaling;
    float ScalingSquared=Scaling*Scaling;
    FourthMomentOffset*=ScalingSquared*ScalingSquared;
    // Prepare a few quantities that will be needed repeatedly
    vec3 InvVals = 1.0 / vec3( 1.0f-Weight, Weight, FourthMomentOffset );

    //  handle root0
    float w0RootWeight = Compute4MomentNonLinearRootWeight(
        NormalizedIntervalEnd,
        InvVals );
    if(IntervalEnd<=Support.x)
        return w0RootWeight*OverEstimationWeight;
    if(NormalizedIntervalEnd>=1.0f)
        return MomentMad( OverEstimationWeight - 1.0, w0RootWeight, 1.0 );

    //  handle root1
    float q=-FourthMomentOffset*InvVals.x/NormalizedIntervalEnd;
    float pHalf=MomentMad(-0.5f*q,MomentMad(NormalizedIntervalEnd,-InvVals.y,1.0f)/(1.0 - NormalizedIntervalEnd),-0.5f);
    float Root=-pHalf-sqrt(MomentMad(pHalf,pHalf,-q));
    float w1RootWeight = Compute4MomentNonLinearRootWeight(
        Root,
        InvVals );
    return w1RootWeight + w0RootWeight*OverEstimationWeight;
}
```

REFERENCES

Morgan McGuire and Louis Bavoil. 2013. Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (18 December 2013), 122–141. http://jcgt.org/published/0002/02/09/

Christoph Peters. 2017. Non-Linearly Quantized Moment Shadow Maps. In *Proceedings of the 9th Conference on High-Performance Graphics (HPG '17)*. ACM. https://doi.org/10.1145/3105762.3105775

Christoph Peters and Reinhard Klein. 2015. Moment Shadow Mapping. In *Proceedings of the 19th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (i3D '15)*. ACM, New York, NY, USA, 7–14. https://doi.org/10.1145/2699276.2699277

Christoph Peters, Cedrick Münstermann, Nico Wetzstein, and Reinhard Klein. 2017. Improved Moment Shadow Maps for Translucent Occluders, Soft Shadows and Single Scattering. *Journal of Computer Graphics Techniques (JCGT)* 6, 1 (30 March 2017), 17–67. http://jcgt.org/published/0006/01/03/