Pacific Graphics 2020
E. Eisemann, A. Jacobson, and F.-L Zhang
(Guest Editors)

Volume 39 (2020), Number 7

Automatic Band-Limited Approximation of Shaders Using Mean-Variance Statistics in Clamped Domain Supplemental Material

Shi Li[1], Rui Wang[1 †], Yuchi Huo[1], Wenting Zheng[1], Wei Hua[2,1], Hujun Bao[1]

[1] State Key Lab of CAD&CG, Zhejiang University, [2]Zhejiang Lab

1. Derivation of Mean-Variance Statistics for Example Code

The Example Codes in Figure 1 (original paper) contain two functions, $x1 = -x^2$ and $x2 = \exp(x1)$. We assume that $X \sim N(\mu_X, \sigma_X^2)$, μ_{X1} and σ_{X1}^2 can be computed through Equation 1 and 2:

$$\mu_{X1} = \int_{-\infty}^{\infty} -x^2 G(\mu_X, \sigma_X^2)dx = -(\mu_X^2 + \sigma_X^2) \quad (1)$$

$$\sigma_{X1}^2 = \int_{-\infty}^{\infty} x^4 G(\mu_X, \sigma_X^2)dx - \mu_{X1} * \mu_{X1} = 4\mu_X^2\sigma_X^2 + 2\sigma_X^4 \quad (2)$$

Subsequently, we assume $X1 \sim N(\mu_{X1}, \sigma_{X1}^2)$. With the similar process, we compute μ_{X2} with μ_{X1} and σ_{X1}^2 as follows:

$$\mu_{X2} = \int_{-\infty}^{\infty} \exp(x)G(\mu_{X1}, \sigma_{X1}^2)dx = \exp(\mu_{X1} + \frac{1}{2}\sigma_{X1}^2) \quad (3)$$

Finally, the smoothed output is $\mu_{X2} = \exp(-\mu_X^2 - \sigma_X^2 + 2\mu_X^2\sigma_X^2 + \sigma_X^4)$.

2. A Proof of the Probability Density Function Approximation for a Single Composition

Let us consider $f(x)$ is a composition of functions f_1 and f_2 as $f(x) = f_2(f_1(x))$. We then break the composition into two sub-parts, $x_1 = f_1(x_0)$ and $x_2 = f_2(x_1)$, and derive the approximation from these two sub-parts. For the variable X_1 where the current domain is $[x_{0a}^o, x_{0b}^o]$, mean (μ_{X_1}) and variance ($\sigma_{X_1}^2$) could be obtained through Equation 3 and 4 in the paper. If a probability density function $p(X_1)$ for the variable X_1 in the current domain satisfies the following equations: $\int_{x_{0a}^o}^{x_{0b}^o} p(X_1)dX_1 = 1$, $\int_{x_{0a}^o}^{x_{0b}^o} X_1 p(X_1)dX_1 = \mu_{X_1}$, $\int_{x_{0a}^o}^{x_{0b}^o} {X_1}^2 p(X_1)dX_1 = \mu_{X_1} + \sigma_{X_1}^2$, then the composition rules still hold when $\sigma \to 0$.

A second-order Taylor expansion is introduced to estimate $f_2(X_1)$ around the mean of the variable X_1:

$$f_2(X_1) \approx f_2(\mu_{X_1}) + \nabla f_2(\mu_{X_1})(X_1 - \mu_{X_1}) + \tfrac{1}{2!}\nabla^2 f_2(\mu_{X_1})(X_1 - \mu_{X_1})^2$$

Then $\overline{f}_2(X_1)$ could be computed as:

$$\begin{aligned}\int_{x_{0a}^o}^{x_{0b}^o} f_2(X_1)p(X_1)dX_1 &\approx \int_{x_{0a}^o}^{x_{0b}^o} f_2(\mu_{X_1})p(X_1)dX_1 + \\ &\int_{x_{0a}^o}^{x_{0b}^o} \nabla f_2(\mu_{X_1})(X_1 - \mu_{X_1})p(X_1)dX_1 + \\ &\int_{x_{0a}^o}^{x_{0b}^o} \frac{1}{2!}\nabla^2 f_2(\mu_{X_1})(X_1 - \mu_{X_1})^2 p(X_1)dX_1 \\ &= f_2(X_1) + \frac{1}{2!}\sigma_{X_1}^2 \nabla^2 f_2(X_1)\end{aligned} \quad (4)$$

With Equation 4, it can be concluded that it is accurate up to the second-order in the assumption of a second-order Taylor expansion.

$$\mu_{X_1} = f_1(\mu) + \frac{1}{2}\sigma^2\nabla^2 f_1(\mu)$$

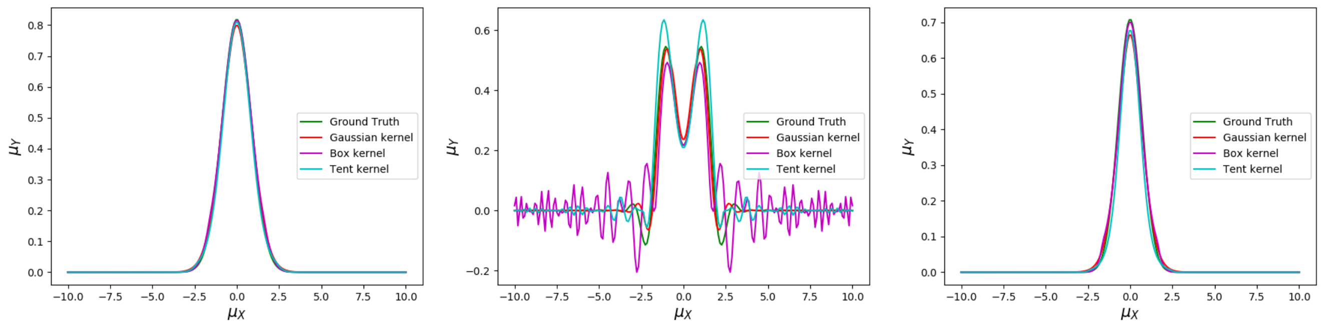
$$\sigma_{X_1}^2 = \sigma^2(\nabla^2 f_1)^2$$

$$\begin{aligned}\mu_{X_2} = f(\mu) + \frac{1}{2}\sigma^2(\nabla^2 f_2(\nabla f_1)^2 \\ + \nabla f_2 \nabla^2 f_1) + O(\sigma^4)\end{aligned} \quad (5)$$

Subsequently, $\overline{f}(\mu, \sigma^2) = f(\mu) + \frac{1}{2}\sigma^2(\nabla^2 f_2(\nabla f_1)^2 + \nabla f_2 \nabla^2 f_1)$. Therefore, the new probability density function also supports second order approximation for a single composition. As shown in Figure 1, Gaussian kernel, box kernel and tent kernel in the current domain are used in the $y = \exp(-x^2)$, $y = \sin(x^2)$, $y = \exp(-2x^2)$ where $\sigma_X^2 = 0.25$. The results verify that the Gaussian function gives a closer approximation to the ground truth for small σ_X. The box kernel may produce more errors but provide a closed-form expression.

3. Renormalized Adaptive Kernels for Box Function and Tent Function

At first, an analytic expression for compactly support kernels can be directly obtained from Equation 5 - 7 in the paper, such as box functions or tent functions.



(a) $y = \exp(-x^2)$ (b) $y = \sin(x^2)$ (c) $y = \exp(-2x^2)$

Figure 1: ***Comparison of approximation*** *the convolution of different simple function compositions with box kernels, tent kernels and Gaussian kernels in clamped domain, gt-integral used the trapezoidal integration method to calculate the convolution.*

For a box function with mean (μ_Y) and variance (σ_Y), the origin probability density function is shown as:

$$f_Y(y) = \begin{cases} 0 & y < \mu_Y - \sqrt{3}\sigma_Y \\ \dfrac{1}{2\sqrt{3}\sigma_Y} & \mu_Y - \sqrt{3}\sigma_Y \leq y \leq \mu_Y + \sqrt{3}\sigma_Y \\ 0 & y > \mu_Y + \sqrt{3}\sigma_Y \end{cases} \quad (6)$$

The renormalized box kernel in the clamped domain $[a^o, b^o]$ can be shown as follows:

$$f_Y(y) = \begin{cases} 0 & a^o \leq y < \mu_Y - Y_1 \\ \dfrac{Y_2}{Y_1^2 + Y_1 Y_2} & \mu_Y - Y_1 \leq y < \mu_Y \\ \dfrac{Y_1}{Y_2^2 + Y_1 Y_2} & \mu_Y \leq y < \mu_Y + Y_2 \\ 0 & \mu_Y + Y_2 \leq y \leq b^o \end{cases} \quad (7)$$

where Y_1 and Y_2 are related with the clamped domain. They must meet $Y_1 Y_2 = 3\sigma_Y^2$ and $[\mu_Y - Y_1, \mu_Y + Y_2]$ must be in the clamped domain $[a^o, b^o]$. Therefore, Y_1, Y_2 should be computed as follows:

$$Y_1 = \max\{\min\{(\mu_Y - a^o), \sqrt{3}\sigma_Y\}, \frac{3\sigma_Y^2}{Y_2}\} \quad (8)$$

$$Y_2 = \max\{\min\{(b^o - \mu_Y), \sqrt{3}\sigma_Y\}, \frac{3\sigma_Y^2}{Y_1}\} \quad (9)$$

For a tent function with mean (μ_Y) and variance (σ_Y), the origin probability density function is shown as:

$$f_Y(y) = \begin{cases} 0 & y < \mu_Y - \sqrt{6}\sigma_Y \\ \dfrac{1}{6\sigma_Y^2}(x - \mu_Y + \sqrt{6}\sigma_Y) & \mu_Y - \sqrt{6}\sigma_Y \leq y < \mu_Y \\ -\dfrac{1}{6\sigma_Y^2}(x - \mu_Y - \sqrt{6}\sigma_Y) & \mu_Y \leq y < \mu_Y + \sqrt{6}\sigma_Y \\ 0 & y > \mu_Y + \sqrt{6}\sigma_Y \end{cases} \quad (10)$$

The renormalized tent kernel in the clamped domain $[a^o, b^o]$ is computed as follows:

$$f_Y(y) = \begin{cases} 0 & a^o \leq y < \mu_Y - Y3 \\ \dfrac{2Y_4}{Y_3 Y_3 (Y_4 + Y_3)}(x - \mu_Y + Y_3) & \mu_Y - Y_3 \leq y < \mu_Y \\ -\dfrac{2Y_3}{Y_4 Y_4 (Y_4 + Y_3)}(x - \mu_Y - Y_4) & \mu_Y \leq y < \mu_Y + Y_4 \\ 0 & \mu_Y + Y_4 \leq y \leq b^o \end{cases} \quad (11)$$

where Y_3 and Y_4 are related with the clamped domain. They must meet $Y_3 Y_4 = 6\sigma_Y^2$ and $[\mu_Y - Y_3, \mu_Y + Y_4]$ must be in the clamped domain $[a^o, b^o]$. Therefore, Y_3, Y_4 could be computed as follows:

$$Y_3 = \max\{\min\{(\mu_Y - a^o), \sqrt{6}\sigma_Y\}, \frac{6\sigma_Y^2}{Y_4}\} \quad (12)$$

$$Y_4 = \max\{\min\{(b^o - \mu_Y), \sqrt{6}\sigma_Y\}, \frac{6\sigma_Y^2}{Y_3}\} \quad (13)$$

Algorithm 1 Automatically genarate a band-limted shader variant

```
Input:
1: The set of nodes for shaders, $N_n$;
2: The chosen rule for each node, $R_n$;
Output:
3: A band-limted shader variant, $newshader$;
4:
5: procedure RULE($R, Node$)
6:     PreNode = node->prenode;
7:     switch $R$ do
8:         case 0
9:             str = Node.name + "_mean = " + OutputMeanFunString(Node.funname, Node.domain, PreNode.mean, PreNode.var) + ";\n"
10:            str += Node's name + "_var = " + OutputVarFunString(Node.funname, Node.domain, PreNode.mean, PreNode.var)+ " - " + Node.name + "_mean *" + Node.name + "_mean;\n"
11:            return str
12:        case 1
13:            str = Node.name + "_mean = " + OutputMeanFunString(Node.funname, Node.domain, PreNode.mean, PreNode.var) + ";\n"
14:            str += Node's name + "_var = " + OutputVarFunString(Node.funname, Node.domain, PreNode.mean, PreNode.var)+ " - " + Node.name + "_mean *" + Node.name + "_mean;\n"
15:            str += "Rule1(" + Node.name + "_mean," + Node.name + "_var," + Node.range + ");\n"
16:            return str
17:        case 2
18:            str = Node.name + "_mean = " + OutputRule2String(Node.funname, Node.domain, PreNode.mean, PreNode.var) + ";\n"
19:            str += Node's name + "_var = " + "1.0 / (" + Node.funname + "_p(" + Node.name + "_mean + ") * "+ Node.funname + "_p(" + Node.name + "_mean)*2*PI;\n"
20:            str += Node's name + "_var = " + Node's name + "_var *" + Node's name + "_var;\n"
21:            return str
22:        case 3
23:            str += "delta_0 =" + Node.function + "(Node.mean)" + ";\n"
24:            str += "delta_1 =" Diff(Node) + ";\n"
25:            str += "delta_2 =" Diff_2(Node) + ";\n"
26:            str += Node.name + "_mean = Rule3(delta_0, delta_1, delta_2," + Node.domain + PreNode.mean + PreNode.var + ");\n"
27:            str += "delta_3 = delta_0 * delta_0;\n"
28:            str += "delta_4 = 2 * delta_0 * delta_1;\n"
29:            str += "delta_5 = 2 * delta_1 * delta_1 + 2 * delta_0 * delta_2;\n"
30:            str += Node's name + "_var = Rule3(delta_3, delta_4, delta_5," + Node.domain + PreNode.mean + PreNode.var + ");\n"
31:            return str
32: end procedure
33:
34: procedure GENERATESHADERCODE()
35:     $newshader$ = template header codes
36:     for $i = 0; i < n; i++$ do
37:         $str$ = RULE($R_i$, $N_i$)
38:         $newshader$ += $str$
39:     end for
40:     $newshader$ += template return codes
41:     return $newshader$
42: end procedure
```

4. Automatically genarate a band-limted shader variant

An algorithm with more details to generate one shader variant is shown in Algorithm 1. With the AST of the shader and one rule, we generate expressions of the mean and variance for each node in order. After the shader variant is generated, we estimate the quality and performance of this variant.

4.1. A Simple Shader Example

Listing 1 shows the example shader code to implement the function $f(x) = \exp(-x^2)$, "pow_2" is a square function.

```
float main(float x){
float x1 = pow_2(x);
float x2 = -x1;
float x3 = exp(x2);
return x3;
}
```

Listing 1: *Original Shader Program.*

The following three subsections provide three code snippets after applying three rules on the example code, which are generated by Algorithm 1 with one rule. The case i $(i = 1,2,3)$ in Algorithm 1 produces the code snippets with the chosen Rule #$i(i = 1,2,3)$ respectively.

4.2. Rule #1 Example Code

Listing 2 describes that we apply Rule #1 to the first function $x1 = pow\_2(x)$. Our approximation takes the domain and range into consideration. At first, we compute the domain and range of each node, containing in the autogenerated codes. Then function "OutputMeanFunString" and "OutputVarFunString" in Line 9 and 10 of Algorithm 1 output code snippets about how to calculate mean and variance, which are depicted as red lines in Listing 2. We will add a suffix "_cov" or "_cov2" to each name of *atomic* function to represent the convolution of $f(x)$ and $f^2(x)$ with a probability density function of the variable x.

For example, the "pow_2" function will be output "pow_2_cov" and "pow_2_cov2" to calculate the convolution. The first two parameters in each convolution function are the mean and variance of the input variable x, the last four parameters denote the clamped domain, we use two flags to show infinity. All the convolution of *atomic* functions have been implemented in the header file.

```
1  float main(float x_mean, float x_var){
2  float x1_mean = pow_2_cov(x_mean, x_var,
       true, 0.0, true, 0.0);
3  float x1_var = pow_2_cov2(x_mean, x_var, true,
    0.0, true, 0.0) - x1_mean * x1_mean;
4  Rule1(x1_mean, x1_var, false, 0, true, 0);
5  float x2_mean = -x1_mean;
6  float x2_var = x1_var;
7  float x3_mean = exp_cov(x3_mean, x3_var, true,
       0.0, false, 0.0);
8  return x3_mean;
9  }
```

Listing 2: *Rule #1 Shader Program.*

Listing 3 shows a numerical method that achieves function "Rule1" in Line 4 of the Listing 2. The purpose of this method is to renormalize x1_mean and x1_var in the clamp domain. The function "norm" gives the result calculated by Equation 8 in the full paper, "UpdateMeanVar" and "UpdateStep" is to update the mean, variance and steps in each iteration. Moreover, we can precompute and store the renormalized values into a lookup table to improve performance.

```
void Rule1(float& mean, float& var, bool leftflag
    , float leftdomain, bool rightflag, float
    rightdomain){
float oldmean = mean, oldvar = var, newmean =
    mean, newvar = var;
float stepmean, stepvar;
float value = norm(newmean, newvar, oldmean,
    oldvar, leftflag, leftdomain, rightflag,
    rightdomain);
for (int iter = 0; iter < 5; iter++){
   UpdateMeanVar(newmean, newvar, stepmean,
       stepvar);
   float tmpvalue = norm(newmean, newvar, oldmean
       , oldvar, leftflag, leftdomain, rightflag,
       rightdomain);
   if(tmpvalue > vaule){
     value = tmpvalue, mean=newmean, var=newvar;}
   else{
   UpdateStep(stepmean, stepvar);
   }
}
}
```

Listing 3: *Implementation of Function "Rule1"*

Listing 4 provides an implementation about the convolution of "exp" function with a Gaussian kernel. We have implemented the convolution of all *atomic* functions.

```
float exp_cov(float mean, doube var, bool
    leftflag, float leftdomain, bool rightflag,
    float rightdomain){
float ret = 0.0;
if (leftflag && rightflag)
{return exp(mean + 0.5 * var);}
else if (leftflag)
{ret =  (0.5 * exp(mean + 0.5 * var) - 0.5 * exp(
    mean + 0.5 * var) * erf((mean + var -
    rightdomain) / sqrt(2 * var)));}
else if (rightflag)
{ret = (0.5 * exp(mean + 0.5 * var) + 0.5 * exp(
    mean + 0.5 * var) * erf((mean + var -
    leftdomain) / sqrt(2 * var)));}
else
{ret =  (-0.5 * exp(mean + 0.5 * var) * erf((mean
     + var - rightdomain) / sqrt(2 * var)) + 0.5
    * exp(mean + 0.5 * var) * erf((mean + var -
    leftdomain) / sqrt(2 * var)));}
return ret;
}
```

Listing 4: *Convolution of exp function with Gaussian function in the clamped domain.*

4.3. Rule #2 Example Code

Listing 5 denotes that we apply Rule #2 to the first function $x1 = pow\_2(x)$. The domain and range are the same as Rule #1. But we add two new functions to describe Equation 11 and 12 in the paper. The function "OutputRule2String" in Line 18 of Algorithm 1 generates blue code snippets shown in Listing 5.

```
1   float main(float x_mean, float x_var){
2   float x1_mean = Rule2("pow_2", x_mean, x_var,
3   true, 0.0,true, 0.0);
4   float x1_var = 1.0 / (pow_2_p(x1_mean, x_mean,
        x_var) * pow_2_p(x1_mean, x_mean, x_var) * 2
        * PI);
5   x1_var = x1_var * x1_var;
6   float x2_mean = -x1_mean;
7   float x2_var = x1_var;
8   float x3_mean = exp_cov(x2_mean, x2_var, true,
        0.0, false, 0.0);
9   return x3_mean;
10  }
```

Listing 5: *Rule #2 Shader Program.*

Listing 6 provides parts of implementations about function "Rule2" in Line 2 and "pow_2_p" in Line 4 of Listing 5.

```
1   float pow_2_p(float x, float mean, float var)
2   {
3   float ret = 1.0 / sqrt(2 * M_PI * var) * exp(-(
        sqrt(x) - mean) * (sqrt(x) - mean) / 2.0 /
        var) * 0.5 / sqrt(x);
4   return ret;
5   }
6   float Rule2(std::string functionname, float mean,
         float var, bool leftflag, float leftdomain,
        bool rightflag, float rightdomain)
7   {
8   switch(functionname)
9   {
10  case exp:
11  {
12      float ret = exp(mean - var);
13      return ret;
14  }
15  case pow_2:
16  {
17      if (mean - 4 * var >= 0)
18      {
19        float ret1 = (mean - sqrt(mean - 4 * var))
              / 2.0;
20        float ret2 = (mean + sqrt(mean - 4 * var))
              / 2.0;
21        ret1 = ret1 * ret1;
22        ret2 = ret2 * ret2;
23        if (pow_2_p(ret1, mean, var) >= pow_2_p(
              ret2, mean, var))
24           return ret1;
25        else
26           return ret2;
27      }
28      else
29      {
30          return 0.0;
31      }
32  }
33  ...
34  }
35  }
```

Listing 6: *Implementation of Function "Rule2"*

4.4. Rule #3 Example Code

Finally, we apply Rule #3 to the whole function. Chain rules are introduced to generate the derivative of multiple *atomic* functions. The "Diff" and "Diff_2" functions generate blue codes shown in Listing 7 that describe the first and second-order derivative of functions.

```
1   float main(float x_mean, float x_var){
2   float t2 = x_mean;
3   float t1 = t2 * t2;
4   float t0 = -t1;
5   float delta_0 = exp(t0);
6   float delta_1 = exp(t0) * (-1) * 2 * t2;
7   float delta_2 = exp(t0) * (-1) * (-1) * 2 * 2
8   * t2 * t2 + exp(t0) * 0 * 2 * t2 + exp(t0) * (-1)
9   * 0 * t2 +exp(t0) * -1 * 2;
10  float x3_mean = Rule3(delta_0, delta_1, delta_2,
        x_mean, x_var, true, 0, true, 0);
11  return x3_mean;
12  }
```

Listing 7: *Rule #3 Shader Program.*

Listing 8 provides the implementations of the function "Rule3" in Line 8 of Listing 7, which achieves Equation 17-19 in the full paper.

```
1   float Rule3(float delta_0, float delta_1, float
        delta_2, float mean, float var, bool leftflag
        , float leftdomain, bool rightflag, float
        rightdomain)
2   {
3   float a = delta_0 * cov_c(mean, var, leftflag,
        leftdomain, rightflag, rightdomain);
4   float ret = 0.0;
5   if (leftflag && rightflag)
6   {
7   float b = 0.0;
8   float c = var;
9   ret = a + delta_1 * b + delta_2 * c;
10  }
11  else if (leftflag)
12  {
13  float b = - sqrt(var / 2 / M_PI) * exp(-(
        rightdomain - mean) * (rightdomain - mean) /
        2 / var);
14  float c = sqrt(var / 2 / M_PI) * (mean -
        rightdomain) * exp(-(rightdomain - mean) * (
        rightdomain - mean) / 2 / var) - 0.5 * var *
        erf((mean - rightdomain) / sqrt(2 * var)) +
        0.5;
ret = a + delta_1 *  b + delta_2 * c;
}
else if (rightflag)
{
float b = sqrt(var / 2 / M_PI) * exp(-(leftdomain
     - mean) * (leftdomain - mean) / 2 / var);
float c = -sqrt(var / 2 / M_PI) * (mean -
    leftdomain) * exp(-(leftdomain - mean) * (
    leftdomain - mean) / 2 / var) + 0.5 * var *
    erf((mean - leftdomain) / sqrt(2 * var)) +
    0.5;
ret = a + delta_1 * b + delta_2 * c;
}
else
{
float b = sqrt(var / 2 / M_PI) * (exp(-(
    leftdomain - mean) * (leftdomain - mean) / 2
    / var) - exp(-(rightdomain - mean) * (
    rightdomain - mean) / 2 / var));
float c = sqrt(var / 2 / M_PI) * (mean -
    rightdomain) * exp(-(rightdomain - mean) * (
    rightdomain - mean) / 2 / var) - 0.5 * var *
    erf((mean - rightdomain) / sqrt(2 * var)) -
    sqrt(var / 2 / M_PI) * (mean - leftdomain) *
    exp(-(leftdomain - mean) * (leftdomain - mean
    ) / 2 / var) + 0.5 * var * erf((mean -
    leftdomain) / sqrt(2 * var));
ret = a + delta_1 * b + delta_2 * c;
}
return ret;
}
```

Listing 8: *Implementation of Function "Rule3".*

5. More Results

5.1. Complex Procedural Shaders

We demonstrate our method on fifteen complex procedural shaders, which were produced by combining 5 base shaders (*Checkerboard*, *Circles*, *Color Circles*, *Quadratic Sine* and *Zigzag*) with 3 choices for parallax mapping: none, bumps, and ripples. All of them are from Yang and Barnes' code [YB18], but we scale the heights of bumps and ripples about three times.

All shaders are approximated by our method, Yang and Barnes' adaptive Gaussian approximation [YB18] and an ideal Gaussian smoothing. The ideal Gaussian smoothing is the convolution of one shader program with Gaussian function computed by Monte Carlo integration. It excludes the errors brought by the compositions of intermediate variables, thereby can be regarded as the theoretical bound of a smoothed shader with Gaussian function. Results shown in Figure 2-4 validate that our method manages to produce better band-limited results than those from Yang and Barnes' work [YB18].

5.2. Shaders with Textures

To handle shaders with textures, we begin with two simple scenes, as shown in Figure 5. Both two scenes have a planar geometry but with different normal maps and normal distribution functions. On the upper row of Figure 5, we show a bumpy normal map with Beckmann distribution function, and on the lower row, we show results computed from a normal hex map with GGX distribution function. As can be seen, direct sampling fails at producing the correct effect. LEAN and our method both can provide non-linear filtering on normal maps, but our method can be generally applied on different distributions, while LEAN is only suitable for Beckmann distribution. Toksvig's method [Tok05] can also handle different distributions, but our method exhibits the best accuracy compared to the ground truth.

Finally, we test physical-based shading shader with GGX BRDF model and normal maps in Sponza scene, a "CheckBoard" procedural shader is also applied on the floor. Our method can both be capable of generating better smoothness of shader programs as well as handling a broader set of shader programs.

References

[Tok05] Toksvig M.: Mipmapping normal maps. *journal of graphics tools 10*, 3 (2005), 65–71. 6

[YB18] Yang Y., Barnes C.: Approximate program smoothing using mean-variance statistics, with application to procedural shader bandlimiting. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 443–454. 6, 7, 8, 9

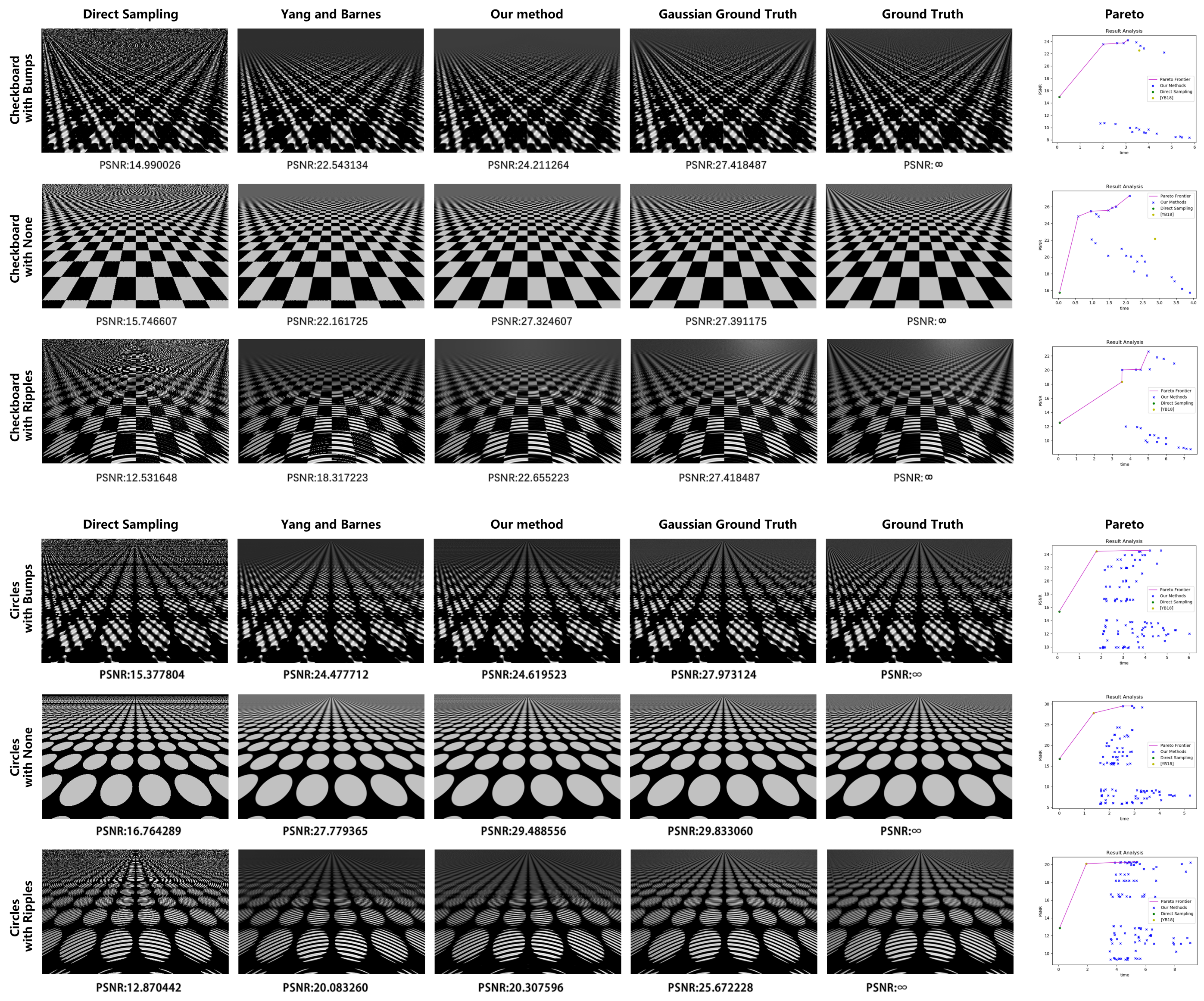


Figure 2: ***Comparison with*** *Direct Sampling, Yang and Barnes [YB18], Our method, Gaussian Ground Truth and Ground Truth under complex procedural shaders.*

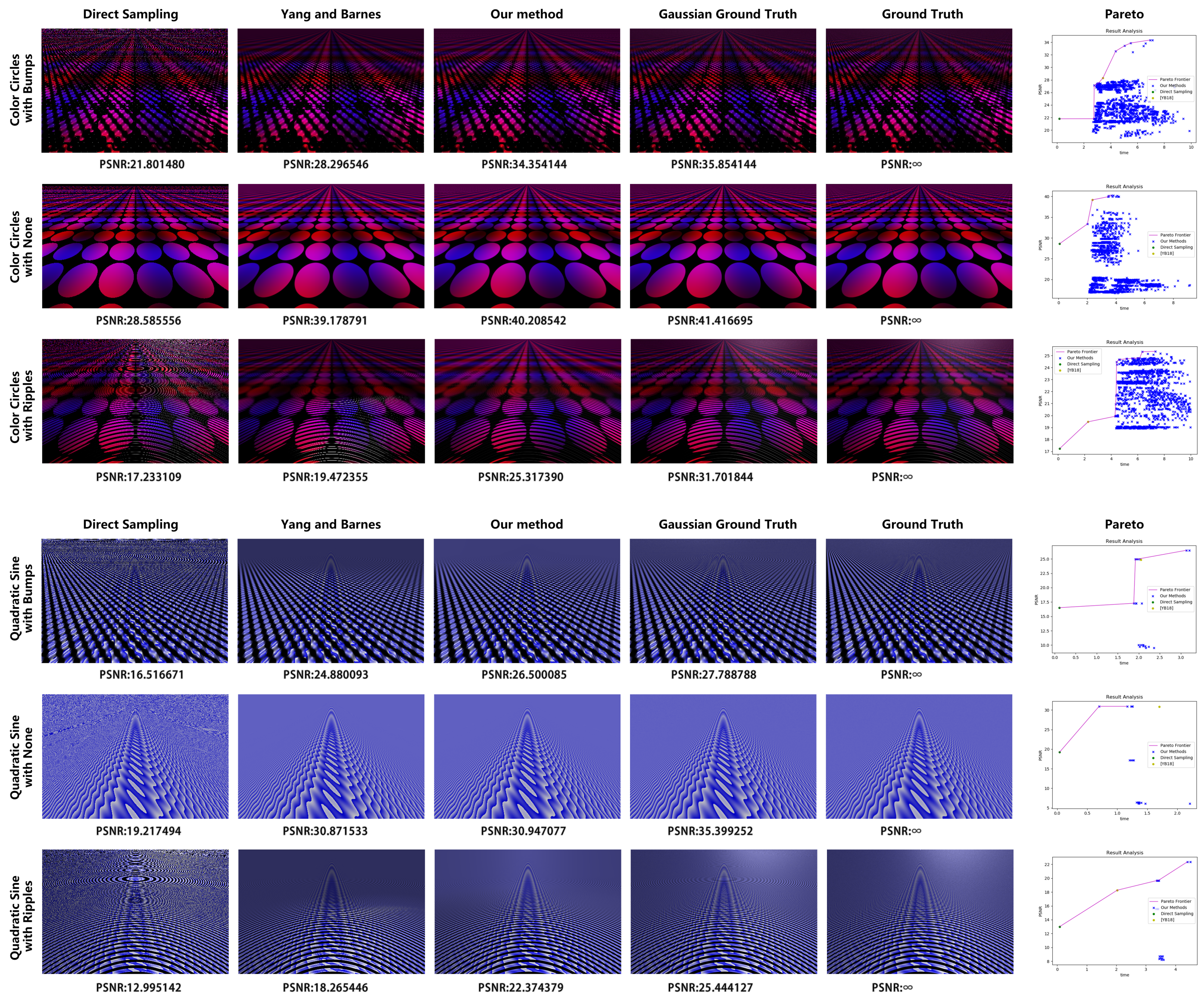


Figure 3: ***Comparison with*** *Direct Sampling, Yang and Barnes [YB18], Our method, Gaussian Ground Truth and Ground Truth under complex procedural shaders.*

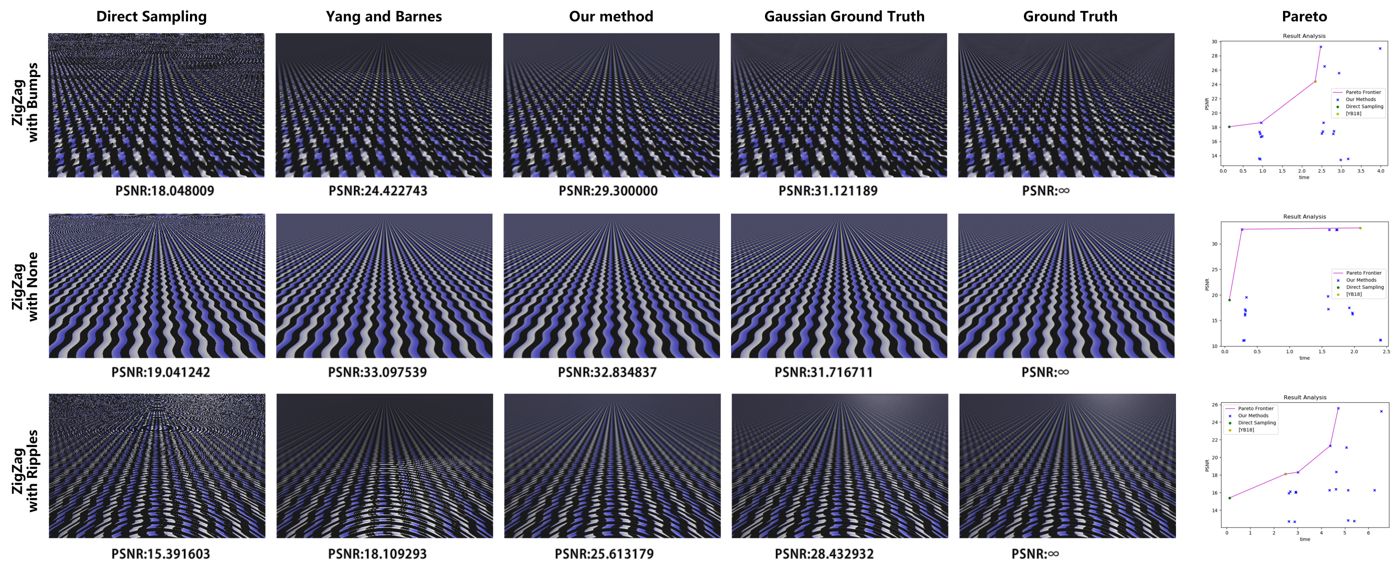
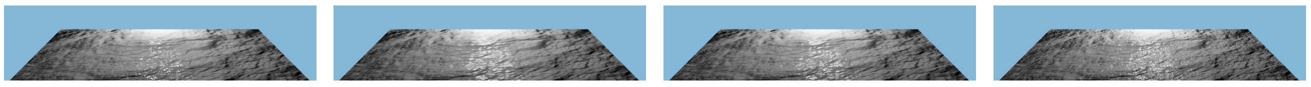
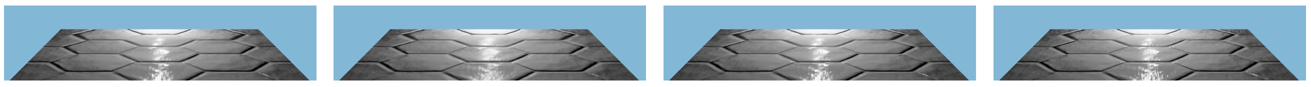


Figure 4: ***Comparison with*** *Direct Sampling, Yang and Barnes [YB18], Our method, Gaussian Ground Truth and Ground Truth under complex procedural shaders.*

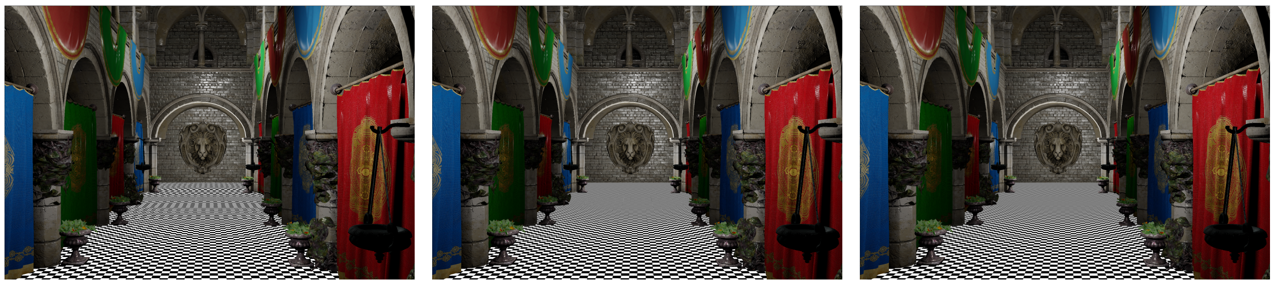


(a) Direct Sampling (b) LEAN (c) Our method (d) Gound truth



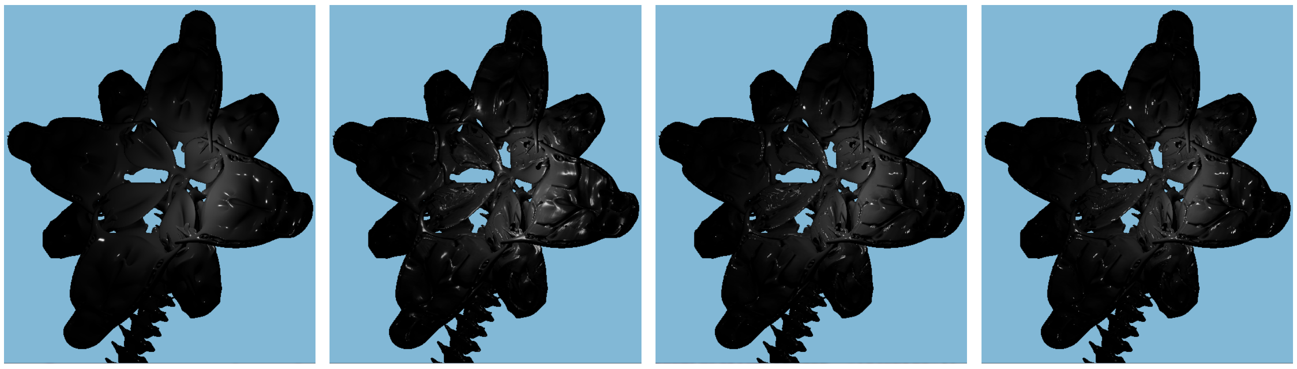
(a) Direct Sampling (b) Toksvig (c) Our method (d) Gound truth

Figure 5: ***Comparison of shading images*** *of two planar surfaces computed by (a) Direct sampling, (b) LEAN/Toksvig, (c) Our method and (d) Ground truth using the physical-based shading shaders with normal maps. The upper and low row show the results using a Beckmann distribution and GGX distribution respectively as normal distribution in BRDF.*



(a) Direct Sampling (**20.9615**) (b) Our method (**27.0221**) (c) Gound truth (∞)

Figure 6: ***Comparison of shading images*** *of the Sponza model computed by (a) Direct sampling, (b) Our method and (d) Ground truth under a physical-based shading shader with GGX BRDF model and normal maps. PSNRs are shown in the bracket.*



(a) Direct Sampling (**0.91184**) (b) Toksvig (**0.94643**) (c) Our method (**0.963251**) (d) Ground Truth (**1.0**)

Figure 7: ***Comparison of shading images*** *of the Desert Rose model computed by (a) Direct sampling, (b) Toksvig, (c) Our method and (d) Ground truth under a physical-based shading shader with GGX BRDF model and normal maps. SSIM values are shown in the bracket.*