IG3DA Project Report: Variance Soft Shadow Mapping

PUTIAN YUAN, Telecom Paris

Variance Soft Shadow Mapping(VSSM) is a technique that fits Variance Shadow Maps(VSM) into Percentage-Closer Soft Shadows(PCSS) framework to render real-time plausible soft shadows. In this report, I will summarize the related works and this paper at first, next describe my personal understanding and implementation details about these techniques, then show my results for each technique and end with a conclusion.

Additional Key Words and Phrases: Soft Shadows, Shadow Maps, Percentage-Closer Filtering, Percentage-Closer Soft Shadows, Summed-Area Table, Variance Shadow Maps

1 SUMMARY

1.1 Shadow Maps

Shadow Maps [Crow 1977] is a two-passes technique to render shadow in 3D scene. In first pass, we put a camera on the light position, setting the camera parameters(near, far, fov and etc according to light source types), and use this light-camera to render the whole scene and store each fragment depth into a texture, which is called Shadow Maps. However, this technique will bring the following issues: z-fighting, no interpolation, shadow acne, peter panning(if bias is too big) and aliasing.

1.2 Percentage-Closer Filtering

Percentage-Closer Filtering(PCF) [Reeves et al. 1987] is a technique to soften the aliasing issue from Shadow Maps. The idea is to use a kernel with a constant size to average the depth-comparison result instead of averaging the depth.

1.3 Percentage-Closer Soft Shadows

Percentage-Closer Soft Shadows(PCSS) [Fernando 2005] is technique based on PCF to render soft shadow with penumbra results. Instead of using a constant size kernel like PCF does, they use a dynamic penumbra size to do the PCF for each fragment, which gives us more plausible soft shadow result.

1.4 Variance Shadow Maps

Variance Shadow Maps(VSM) [Donnelly and Lauritzen 2006] is also a Shadow Maps type technique. However, instead of storing only depth information, VSM stores depth and depth-square into a color-texture which can be interpolated, filtering and mipmap generation. By using depth and depth-square, they use Chebychev's inequality to compute the upper bound of lighting ratio on a surface. It overcomes the depth-interpolation and aliasing issues of traditional Shadow Maps, but bring the light bleeding issue.

This report is submitted as a part of project for Advanced 3D Computer Graphics (IMA904/IG3DA), Telecom Paris.

The original work is introduced by [Yang et al. 2010].

1.5 Summed-Are Table Variance Shadow Maps

Summed-Are Table Variance Shadow Maps(SAT-VSM) [Lauritzen 2007] is a technique based on VSM and Summed-Area Table(SAT) [Crow 1984]. It comes up a new elegant formula for computing $E(x)$ and $E(x^2)$ to eliminate the shadow acne, and also solves the light bleeding issue from VSM. Besides, it integrates SAT into VSM to keep FPS stable when kernel is getting large. What's more, it also shows how to fit VSM into PCSS.

1.6 Variance Soft Shadow Mapping

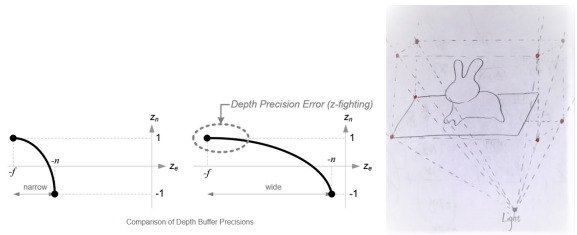
Variance Soft Shadow Mapping(VSSM) [Yang et al. 2010] is a technique similar to SAT-VSM. The main contributions in this paper are: a novel formula for estimating blocker depth and a kernel subdivision scheme. Unfortunately, in my experiment, I could not verify the correctness of the formula and the necessity for the subdivision. In my personal point of view, I think SAT-VSM has already done well.

2 IMPLEMENTATION

In order to compare VSSM with other shadow techniques, I have implemented Shadow Maps, PCF, PCSS, VSM, SAT, and VSSM.

2.1 Shadow Maps

When using Shadow Maps technique, we need to handle the z-fighting issue as figure 1(a) shown. In order to improve the depth precision, I have adopted the idea to create a tight light space from this article [MicroSoft 2020]: I use a tight light frustum to encapsulate the scene bounding box, as figure 1(b) shown. The z-fighting issue now can be mitigated as the distance between near and far plane is closer. As for biasing, it depends on the scene, which means we need to tweak the bias until it removes the obvious shadow acne without causing peter panning issue.



(a) (b)

Fig. 1. left: z-fighting;
Right: Light Tight Space Computation

2.2 PCF

It is very easy to implement PCF in shader: I just iterate all texels of shadow map over a kernel, then average the compared result(the

compared result is binary: either 0 or 1) to get a value varying from 0 to 1.

2.3 PCSS

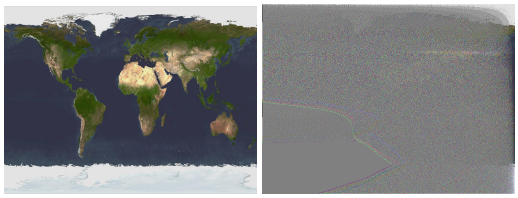
There are three stages in PCSS: compute blocker-depth, penumbra size estimation, filtering over a kernel with penumbra size. At the blocker search step, inside a search region, we need to compare each texel's depth from Shadow Maps with current fragment depth, as a consequence, we can not use SAT technique. This brute-force iteration will slow down the performance as the search kernel size increases. Therefore, I separate the search kernel size and the light size instead of using the light size as PCSS proposed. This allows us to set a very large light size without increasing the search area. At the second stage, I use the same penumbra size estimation formula:

$$\omega_{penumbra} = \frac{\omega_{light}(d_{receiver} - d_{blocker})}{d_{blocker}}$$
$$= \omega_{light}(\frac{d_{receiver}}{d_{blocker}} - 1)$$

, we can see $\frac{d_{receiver}}{d_{blocker}}$ can be a very large number, which can possibly cause penumbra size to become uncontrollable. Hence, I clamp penumbra size into a user-defined range to avoid extreme size.

2.4 SAT-VSM

2.4.1 SAT.

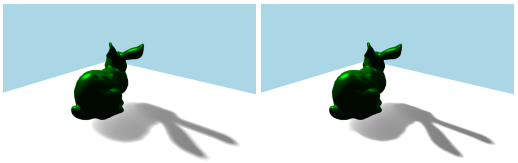


(a) (b)

Fig. 2. left: Original image;
Right: Reconstructed image from 16-bits SAT

In order to implement SAT-VSM, at first I followed this paper: Fast Summed-Area Table [Fas 2005] to generate the SAT for variance shadow maps. In my experiment, using 16-bits texture for SAT is not enough, as we can see, the reconstructed image 2(b) has lost most of the information comparing with the original image 2(a). My computer can not support reading 16 texels as the paper proposed but only up to 8 texels at the same time. Therefore, I used a 32-bits texture for SAT and read 8 texels when generating SAT. In order to improve the floating-point precision, I subtracted 0.5 for original input before generating SAT, and compensate this loss during reconstruction. However, in my experiment, it is still suffering the floating-point precision issue, but it's acceptable for most of the case.

2.4.2 VSM with PCSS.



(a) (b)

Fig. 3. left: Projected Depth; Right: Linear Depth

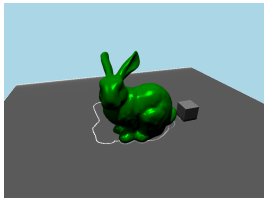
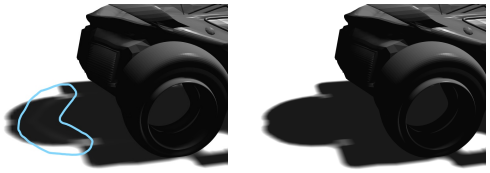


Fig. 4. Light bleeding - basic VSM without biasing



(a) (b)

Fig. 5. left: Light bleeding; Right: Reduce light bleeding

As for generating VSM, I just followed the article [Lauritzen 2007] and implemented it. Using the new formula to compute M2 can eliminate the shadow acne. Instead of storing projected Z-depth and its square value into texture, I choose to store the linear depth: I project the distance between surface point and light source onto light view direction and scale it into range $[0, 1]$ *by using near and far plane of light space. We won't have the z-fighting issue when using this linear depth. The difference between two type depth is shown as figures 3(a) and 3(b). I have also used a 32-bits floating-point RG-color texture, enable its linear interpolation and generate mipmap for it. As for light bleeding issue, it is because VSM only give a upper bound light ratio on a surface point. When the variance is large enough and its contribution to the formula is much bigger than other parts, it will enlarge* P_{max}*. From this observation, light bleeding will occur on the edges, as figure 4 shown. In order to reduce it, I remove the range* $[0, P_{min}]$ *then scale the* $[P_{min}, 1]$ *into range* $[0, 1]$*, as SAT-VSM proposed. Its side effect is darkening the shadow. We can see the compared results from figures 5(a) and 5(b).*

There is one thing important in my implementation about retrieving the moment from SAT. Because SAT stores the sum of an area, it can not be interpolated and we can only read it texel by texel. Using "texture()" function in shader is not correct, we should only use "texelFetch()"

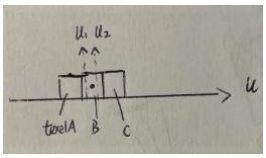


Fig. 6. Neighbor texels along with u direction

function. However, "texelFetch()" only accepts texel space coordinate, and it is a integer value. In our fragment shader, we will have a floating point UV coordinate. Therefore, we need to convert this UV coordinate to texel coordinate. But as we know, this UV coordinate usually does not locate exactly at the center of texel. It means getting one texel coordinate from UV is not enough(it becomes returning the nearest texel). For better result, We need to get its four neighbor texels and do the bilinear interpolation by ourselves. Here are my implementation: for each UV, I compute its surrounding four texels. For example, as figure 6 shown, the neighbor texels of u_1 are texel A and B, while the neighbor texels of u_2 are B and C. Once we have four texels, I use each texel to get its moment, then using the distance to these four texels as weights to get their bilinear interpolated moment.

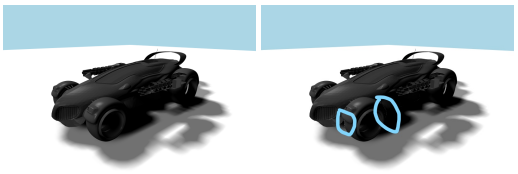
As for fitting PCSS into SAT-VSM, first I used user-defined search area size to get an averaged blocker depth, then used this blocker depth to estimate the penumbra size, finally used this penumbra size as kernel size to get the M1, M2 from VSM to compute the upper bound of lighting ratio. The only difference between using PCSS and not using it is the kernel size. In short, the kernel size for getting moment from VSM is constant(user-defined) if we don't use PCSS, otherwise, it is equal(or proportional) to the penumbra size(dynamic).

2.5 An attempt of separating lighting and shadowing

So far, I have implemented traditional Shadow Maps, PCF, PCSS, SAT, VSM. In my implemenation, all these techniques are combining lighting and shadowing together, which means that I compute the light ratio, ambient, diffuse and specular to get the final color response in one pass, as this formula shown:

$$colorResponse = ambient + lightRatio * (diffuse + specular) \quad (1)$$

By observing this formula1, I started to wonder if we use two passes to separate the computation of light ratio(shadowing) and final color(lighting), then we can use any filter to smooth the light ratio to improve shadow result. However, in my experiment, there is an serious issue: floating-point precision. When I tried to render the light ratio into a texture, it will lose some precision based on texture internal format(8-bits, 16-bits or 32-bits), then reading the light ratio from the texture will cause very obvious error: image-crack. As figure 7(a) shown, it is rendered by using one pass, while figure 7(b) shows the crack error by using two passes. I have done such a test: when using two passes, I simply output any floating-point value in range $[0, 1]$, then I check whether the input still remain the same value in the second pass. As a result, I found those floating-point value which can be represented perfectly in binary format, such as 0, 0.5, 1 and etc, won't lose its precision because it has a terminating representation in binary format. As for those floating-point value with infinite binary representation, it will definitely be truncated. Therefore, it is not a good option to separate the lighting ratio computation and shading. If we really want to do it, there is a possible way to do it, using two integer to represent a floating-point such as $\frac{integer_1}{integer_2}$. This technique is very well known to solve this floating-point issue in frame-based synchronization.



(a) (b)

Fig. 7. left: one pass; right: two passes

2.6 VSSM

VSSM is similar to SAT-VSM with PCSS. Their contribution is to solve the "non-planarity" issue: a novel formula for computing the blocker depth, and kernel subdivision scheme. In my personal point of view, SAT-VSM with PCSS has already done well. According to this paper, they use such a condition:

$$Z_{Avg} \geq d \quad (2)$$

, where Z_{Avg} is the averaged-depth(mean) over a kernel region, and d is current fragment depth. It is true that when the kernel size increase, using this condition to continue our VSM algorithm will misclassify some surfaces which are actually not fully-lit but classified as fully-lit, especially the variance is high. Then they use 4 condition to find those fragments with "non-planarity" issue. Once it has been found, they subdivide the kernel size until the sub-kernel size is small enough. For each small kernel, if it's satisfied "non-planarity" condition, they just simply use brute-force PCF method. If it is a "planar" kernel, they use their novel formula:

$$Z_{Occ} = \frac{Z_{Avg} - P_{max} Z_{unocc}}{1 - P_{max}} \quad (3)$$

to compute the blocker depth(Z_{Occ}. P_{max} is the upper bound computed from VSM theory, and they assume Z_{unocc} is actually current fragment depth for planar situation. However, I could not prove its correctness. By analysing this formula, they only use this formula when $Z_{Avg} < d$(they call it as planar case), then they assume $Z_{unocc} = d$. Now we observe the part $(Z_{Avg} - P_{max} d)$, since we know $Z_{Avg} < d$, we can let $Z_{Avg} = P_0 d$, and P_0 must be less than 1. Now we rewrite the above part to get

$$Z_{Occ} = (P_0 - P_{max})(\frac{d}{1 - P_{max}}) \quad (4)$$

We know P_0 and P_{max} are in range $[0, 1]$, but we can not make sure that $(P_0 - P_{max})$ will return a positive value, which means Z_{Occ} could be a negative value. In my experiment, it did show some negative Z_{Occ}. First I rendered the scene with normal VSSM, as figure 8(a) shown, then I discarded the fragment when Z_{Occ} was negative. As we can see, there are many fragments having negative

Z_{Occ}. In order to analyse this issue, considering now we are shooting light rays on the plane, and some of them are hitting on the back or ear of this bunny. For these fragments, we can easily see, the depth changes very quickly over a tiny area and the variance is very high even the kernel is extreme small(e.g. two texels). Therefore, I think their assumption is not correct, and the following work based on it can be not verified as correct. As for their subdivision, it is a feasible method but it will slow down the performance since they use brute-force subdivision. What's more, if we separate blocker depth search area size and light size as I mentioned above 2.3, we can already have a small enough kernel without subdivision. Personally, due to my limited ability, I am not sure whether my implementation of VSSM is correct or not, but in my experiment, I really could not verify its correctness and necessity.

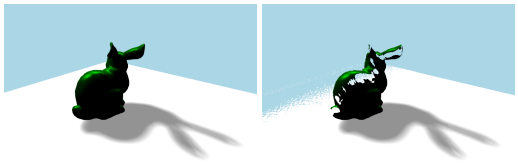


Fig. 8. left: normal VSSM rendering; right: discard fragments when Z_{Occ} is negative

2.7 Code Base

All my implementation is based on my personal render: IceRender. For comparing different technique and effect of different parameters, we can use command to dynamically initialize the scene by loading and changing different scene files or shader files.

3 RESULTS

Here are the results for each technique: (Large and clear images can be found in here.)

Shadow Maps: bunny and cube 9(a), Shadow Maps with PCF: bunny and cube 9(b), Shadow Maps with PCSS: bunny and cube 9(c), Shadow Maps with PCSS: contact object 9(d),

VSM: bunny 9(e), VSM: car 9(f), VSM: mutiple planes 9(g),

VSM with PCSS: bunny 9(h), VSM with PCSS: multiple planes 9(i), VSM with PCSS: contact Object 9(j)

VSSM: bunny 9(k) VSSM: contact Object 9(l)

4 CONCLUSIONS

Shadow Maps can be used to render shadow, but it has many issues. PCF mitigates its aliasing issue but kernel size is constant. PCSS improves PCF by estimating dynamic penumbra size. Both PCF and PCSS will still give box-filtered results. VSM provides an elegant method which allows interpolated depth texture to render soft shadow but bring the light-bleeding issue. SAT-VSM has solved all the issues mentioned before, also combing the SAT to improve the performance when kernel is large. This technique also can be used to fit PCSS to render more plausible soft shadow. VSSM is similar to SAT-VSM. Due to my limited ability, I could not verify the correctness of their main contribution in my implementation, but the subdivision scheme can be useful for future work. Source codes can be found here.

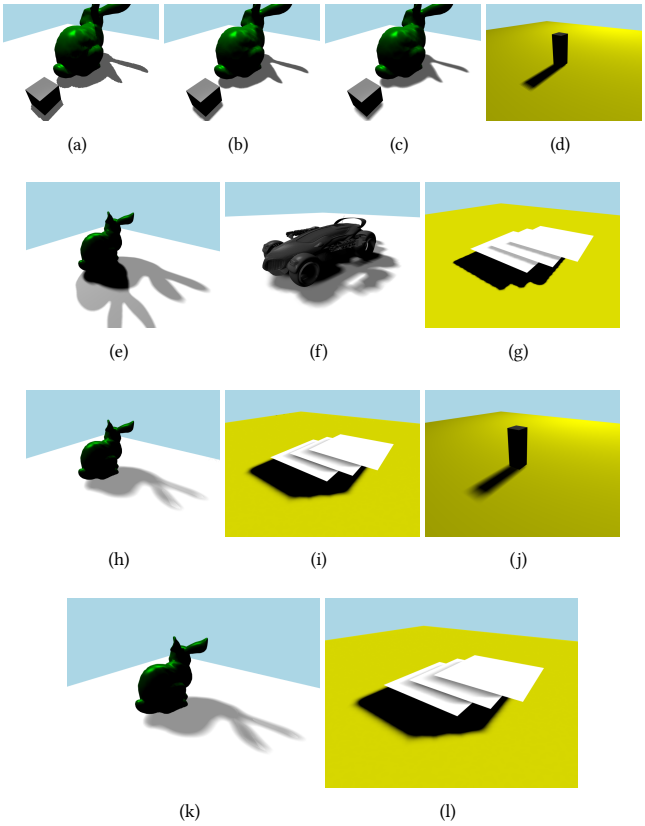


Fig. 9. results

REFERENCES

2005. Fast Summed-Area Table Generation and its Applications. *Computer Graphics Forum* 24, 3 (2005), 547–555. https://doi.org/10.1111/J.1467-8659.2005.00880.X

Franklin C. Crow. 1977. Shadow Algorithms for Computer Graphics. *SIGGRAPH Comput. Graph.* 11, 2 (jul 1977), 242–248. https://doi.org/10.1145/965141.563901

Franklin C. Crow. 1984. Summed-Area Tables for Texture Mapping. *SIGGRAPH Comput. Graph.* 18, 3 (jan 1984), 207–212. https://doi.org/10.1145/964965.808600

William Donnelly and Andrew Lauritzen. 2006. Variance shadow maps. *Proceedings of the Symposium on Interactive 3D Graphics* 2006, 161–165. https://doi.org/10.1145/1111411.1111440

Randima Fernando. 2005. Percentage-closer soft shadows. In *International Conference on Computer Graphics and Interactive Techniques*.

Andrew Lauritzen. 2007. *GPU Gem3: Chapter 8. Summed-Area Variance Shadow Maps.* https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-8-summed-area-variance-shadow-maps

MicroSoft. 2020. *Common Techniques to Improve Shadow Depth Maps.* https://learn.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps

William T. Reeves, David H. Salesin, and Robert L. Cook. 1987. Rendering Antialiased Shadows with Depth Maps. *SIGGRAPH Comput. Graph.* 21, 4 (aug 1987), 283–291. https://doi.org/10.1145/37402.37435

Baoguang Yang, Zhao Dong, Jieqing Feng, Hans-Peter Seidel, and Jan Kautz. 2010. Variance Soft Shadow Mapping. 29, 7 (2010), 2127–2134.