# CODE SAMPLE

## Temporally Stable Conservative Morphological Anti-Aliasing (TSCMAA)

An anti-aliasing technique to provide better temporal stability and performance workaround for virtual reality applications

Sungye Kim

| | |
|---|---|
| **File(s)** | ( the code  sample is attached) |
| **License** | BSD 3-clause "New" or "Revised" License |

## Optimized with…

| | |
|---|---|
| **OS** | (Check all that apply)<br>☐ Android* OS ☐ Apple* iOS ☐ OS X* ☐ Arduino*<br>☐ FreeBSD* ☐ Chrome* OS ☐ Linux* ☐ MeeGo*<br>☐ Windows* (XP*, Vista*, 7) ☒ Windows® 10<br>☐ Windows 8.x ☐ Moblin* ☐ Tizen* ☐ Unix*<br>☐ Wind River* Linux ☐ Wind River Rocket* ☐ Yocto Project* |
| **Hardware** | Intel® HD Graphics 630 |
| **Software** | Microsoft Visual Studio* 2017, Universal Windows* Platform development, Windows 10 SDK 10.0.15063 |
| **Prerequisites** | An understanding of DirectX* 11 is helpful but not essential. |

Seshupriya Alluru, Rebecca David, Matthew Goyder, Anupreet Karla, Yaz Khabiri, Sungye Kim, Pavan Lanka, Filip Strugar, and Kai Xiao contributed to TSCMAA and samples provided in this document.

This article describes Temporally Stable Conservative Morphological Anti-Aliasing (TSCMAA), which was designed and developed as a multisample anti-aliasing (MSAA) alternate technique to provide better temporal stability and performance workaround for virtual reality (VR) applications.

## Introduction

Anti-aliasing (AA) in computer graphics refers to a set of techniques used to overcome aliasing artifacts in the rendered images, which are a byproduct when representing a high-resolution image at a lower resolution (for example, rasterization or sampling). MSAA is the most widely used spatial AA algorithm in various applications but at the cost of performance. Post-processing AA algorithms like Conservative Morphological Anti-Aliasing (CMAA), Fast Approximate Anti-Aliasing (FXAA), and Subpixel Morphological Anti-Aliasing (SMAA) can provide constant performance but suffer from temporal instability. For VR, AA is more critical since the display is closer to the eyes, thus artifacts are more noticeable, and temporal stability becomes a key aspect in providing a good user experience.

This article discusses TSCMAA, which employs an optimized CMAA and integrates temporal accumulation in order to create a temporally stable post-processing AA solution as a MSAA alternate technique without compromising on image quality. TSCMAA is also designed to run efficiently on low- and medium-end graphics processing units (GPUs), such as integrated GPUs, and to be minimally invasive. This makes it acceptable as an MSAA alternate with better temporal stability in a wide range of applications, which include aliasing-prune geometry such as text, patterns, lines, and foliage.

Figure 1 shows a TSCMAA flow that combines a CMAA pass (top) and a TAA pass (bottom). In TSCMAA, CMAA and TAA passes run only for edge candidates identified by edge detection, resulting in fast indirect dispatch of shaders.

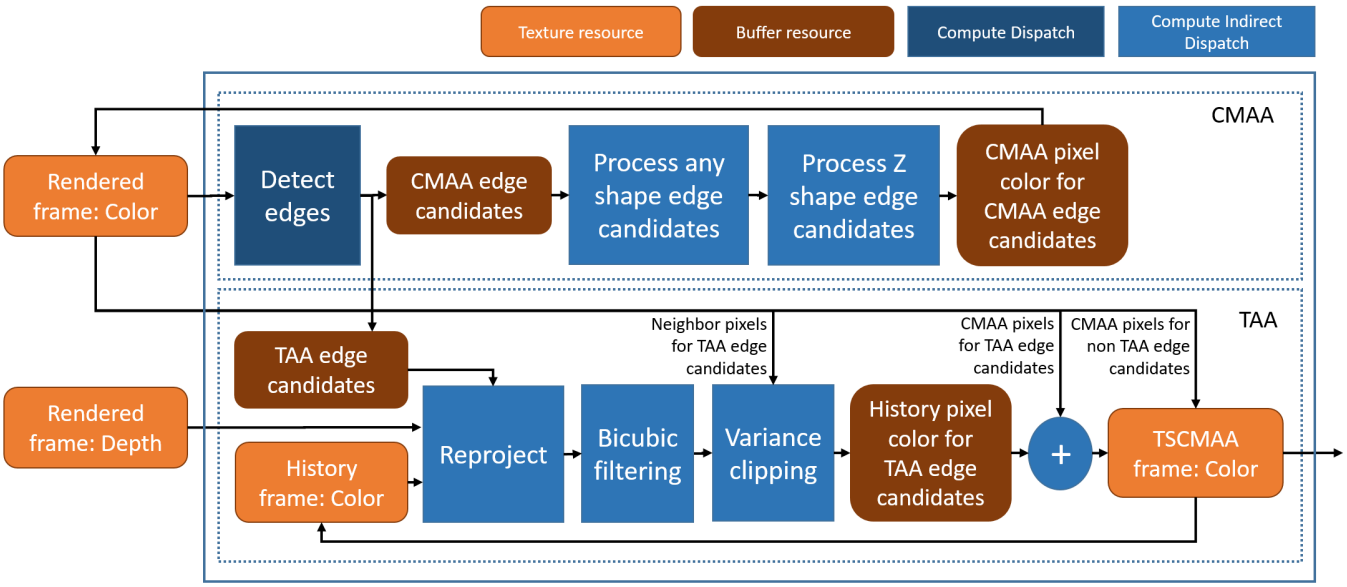


Figure 1. TSCMAA flow.

## Conservative Morphological Anti-Aliasing

CMAA takes a rendered color frame as an input and updates the frame with spatially anti-aliased edge pixels by processing edge candidates (any shape and Z-shape edges). To identify such edge candidates, edge detection uses the luminance difference in color with a given edge threshold. In TSCMAA, a default edge threshold value is 0.045f (1.f/22.f) based on experiments with our sample scene, but it can be adjusted for other scenes as a quality versus performance knob. The details of CMAA are found in the original article.

## Temporal Anti-Aliasing

To reduce temporal aliasing such as shimmering or crawling caused by the motion between frames, TAA blends a current pixel with a history pixel. Then the output of TAA becomes a history frame for the next TAA frame in a feedback loop. In VR, TAA is more crucial due to quite a bit of jitter from head pose changes over time, resulting in rendered frames always containing motion in a head-mounted display (HMD). TSCMAA takes care of both spatial and temporal aliasing by combining CMAA and TAA, and major benefits arise because TSCMAA runs only for edge pixels, resulting in faster pixels that are less blurred overall, compared to the results of other TAA algorithms.

## TAA Edge Candidates

Only a portion of edge candidates from edge detection are considered as TAA edge candidates. In TSCMAA, we use 50 percent of CMAA edge candidates for TAA. This default value is based on experiments with our sample scene but can be adjusted for other scenes as a quality versus performance knob. Figure 2 shows edge detection results with the default edge threshold value for CMAA edge candidates on the left and TAA edge candidates on the right. The TAA pass is applied only to the pixels in TAA edge candidates, resulting in fast TAA and less blurring on non-edge pixels.

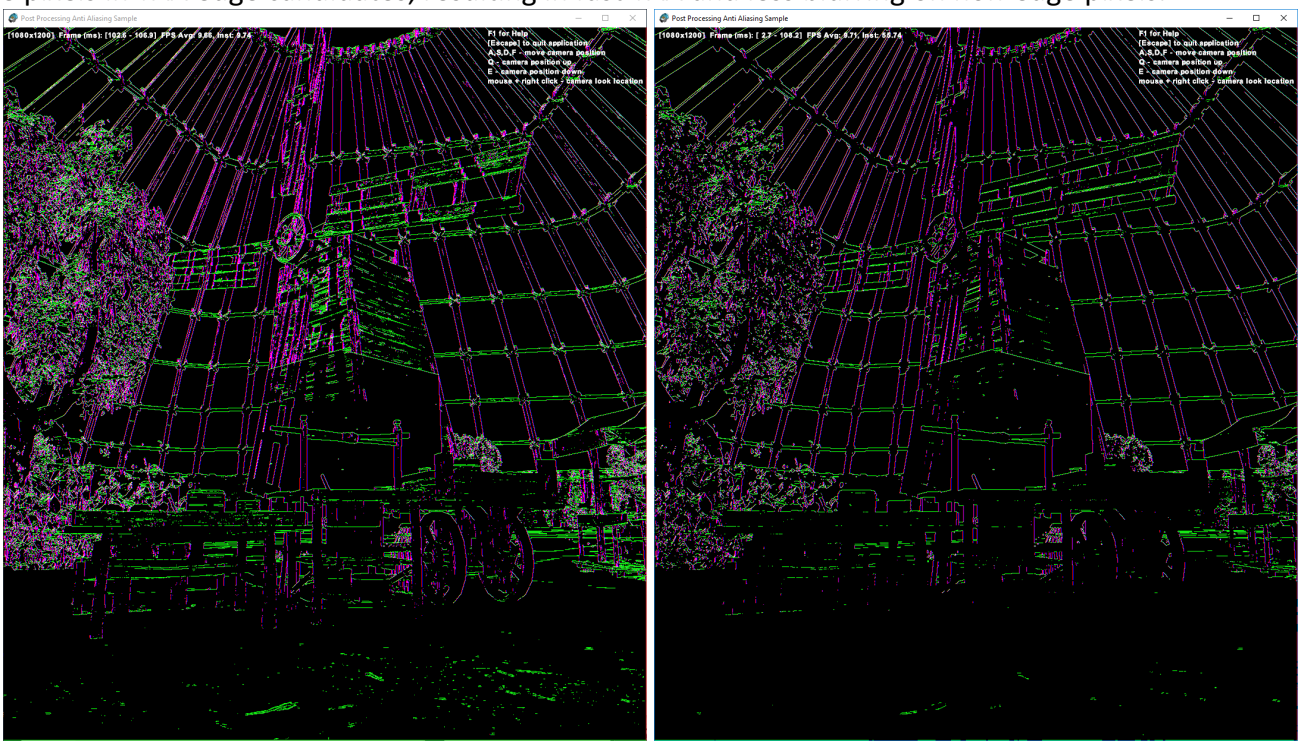


**Figure 2.** Edge candidates for CMAA (left) and TAA (right). TAA edge candidates are 50 percent of the sCMAA edge candidates.

## History Pixel Sampling

The history frame maintains the previous TSCMAA frame so that a CMAA pixel is blended with a history pixel to generate the current TSCMAA frame (see Figure 1). To find a correct history pixel, we reproject a texture coordinate with view and projection of the current rendered frame based on depth. Then a history pixel is sampled with the reprojected texture coordinate using bicubic sampling. TSCMAA employs a five-tap approximation for a Hermite/Catmull-Rom bicubic filter for sharpness-preserving sampling in a history frame but with faster approximation.

## Variance Clipping

When there are moving objects in a scene, reusing stale pixels in the history frame creates a ghosting artifact. A neighborhood color clipping using an axis-aligned bounding box (AABB) has been used as an alternative for expensive color clipping using a convex hull. However color clipping using AABB results in poor quality when a new color clipped by AABB is too far from the original pixel. Many articles discuss the benefits of variance clipping to provide less ghosting artifacts. TSCMAA uses variance clipping in YCoCg space to minimize such a ghosting artifact on moving objects.

## Blending with the CMAA Pixel and TSCMAA History Pixel for TAA Edge Candidates

Since the TAA pass is processed only for TAA edge candidates, blending between CMAA pixels and history pixels is also done only for TAA edge candidates with a blend weight of 0.8f and other non TAA edge pixels are directly taken from CMAA pixels which is same to using a blend weight of 0.f. Then the final blended output generates a TSCMAA frame and becomes a history frame for the next TSCMAA frame as shown in Figure 1.

## Sample Applications

In the TSCMAA sample package attachment, we provide two VR sample applications using the same test scene as shown in Figure 3. One (AASample_WMR) is for Microsoft's Windows Mixed Reality (WMR) HMD and the other (AASample) uses OpenVR* so that it will work on all OpenVR-compatible HMDs. AASample can also run as a desktop application without the "_OPENVR_" preprocessor in project configuration properties.

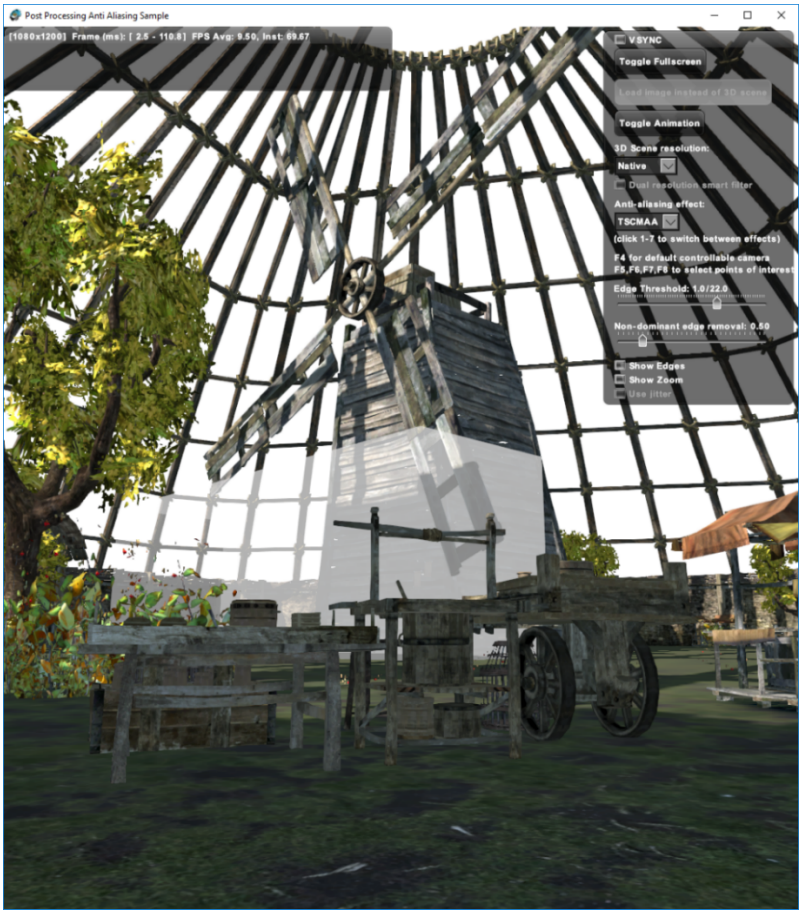


Figure 3. TSCMAA sample scene with lines, foliage, and a transparent object.

## Quality and Performance

Figure 4 shows a quality comparison among MSAA2x, MSAA4x, TSCMAA, and No AA. TSCMAA quality is moderately equivalent to MSAA4x when viewed with an HMD and overall better than MSAA2x with default edge threshold values. Since the TAA pass in TSCMAA is based on temporal accumulation, somehow blurred pixels are inevitable in the final TSCMAA frame. However, we minimize the blurring by accumulating only TAA edge pixels in a history frame and provide better temporal stability. Since we cannot show TAA benefits with static screenshots, this article also provides an attached sample code package with TSCMAA shaders .

Figure 4. Quality comparison among MSAA2x (top left), MSAA4x (top right), TSCMAA (bottom left), and No AA (bottom right).

TSCMAA performance depends on multiple aspects, such as edge count in the scene and render target resolution. For our test scene, TSCMAA shows MSAA4x quality with around a 40 percent cost reduction and is equivalent to MSAA2x performance with default edge threshold values. Here we also show how TSCMAA performance scales as the edge count in the scene and render target resolution change. These experiments were performed with 1280×1280/eye on Intel® HD Graphics 630 system at 1150 MHz.

Since TSCMAA runs only on edge candidates after edge detection, and performance depends on the edge pixel count in the current view. Table 1 shows TSCMAA performance scaling for different edge counts from 53K to 100K pixels. The baseline contains 73K edge pixels, which is from a view shown in Figure 3. Table 1 shows TSCMAA performance changes of 10–15 percent for a 30 percent difference in edge count.

Table 1. TSCMAA performance scaling ratio for different edge count. Baseline (1.0) has 73K edges and we compare with different scene (or views) where edge count is 30 percent less or greater than baseline.

| Scale Factor over Base | 53K Edges | 73K Edges (Base = 1.0) | 100K Edges |
|---|---|---|---|
| Edge count | 0.72 | 1.0 | 1.36 |

| | | | |
|---|---|---|---|
| TSCMAA time | 0.86 | 1.0 | 1.08 |

Table 2 shows TSCMAA performance scaling for 2K×2K render target resolution while maintaining the same edge count. Compared to a baseline with 1280×1280 resolution, a 2K×2K render target contains a 2.56x larger number of pixels, contributing to a TSCMAA performance increase of up to 1.6x. The increase is mostly from an edge detection step which is resolution dependent.

Table 2. TSCMAA performance-scaling ratio for different render target resolution. Baseline (1.0) is 1280×1280 per eye, and we compare with 2K×2K per eye while preserving edge count.

| Scale Factor over Base | 1280×1280/Eye (Base = 1.0) | 2K×2K/Eye |
|---|---|---|
| Pixel count | 1.0 | 2.56 |
| Edge count | 1.0 | 1.0 |
| TSCMAA time | 1.0 | 1.59 |

Table 3 shows the combined contribution from larger render target resolution and thereby the increased number of edge pixels.

Table 3. TSCMAA performance scaling ratio for different render target resolution.

| Scale Factor over Base | 1280×1280/Eye (Base = 1.0) | 2K×2K/Eye |
|---|---|---|
| Pixel count | 1.0 | 2.56 |
| Edge count | 1.0 | 1.91 |
| TSCMAA time | 1.0 | 2.12 |

## Supported Resource Formats

To apply TSCMAA, applications are required to use one of TSCMAA supported resource formats for render target and depth resources. For the render target texture, TSCMAA supports 32- bit RGBA and BGRA typeless formats, such as

- DXGI_FORMAT_R8G8B8A8_TYPELESS
- DXGI_FORMAT_B8G8R8A8_TYPELESS

and creates internal view resources with corresponding UNORM and UNORM_SRGB formats.

- DXGI_FORMAT_R8G8B8A8_UNORM
- DXGI_FORMAT_R8G8B8A8_UNORM_SRGB
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM_SRGB

The 32-bit BGRA format is the only resource format that Microsoft's WMR application uses at the time of this writing.

For depth, TSCMAA supports 24- and 32-bit resource formats with or without a stencil, such as:

- DXGI_FORMAT_D32_FLOAT
- DXGI_FORMAT_D24_UNORM_S8_UINT
- DXGI_FORMAT_D32_FLOAT_S8X24_UINT
- DXGI_FORMAT_R32_TYPELESS
- DXGI_FORMAT_R24G8_TYPELESS
- DXGI_FORMAT_R32G8X24_TYPELESS

### TSCMAA Interface

TSCMAA provides a simple interface to support both standard desktop and VR applications.

`TSCMAA::Create(…)` initializes TSCMAA.

```
HRESULT TSCMAA::Create(ID3D11Device * pDevice,
                       DXGI_FORMAT format,
                       int width,
                       int height,
                       unsigned int numEyes = 1);
```

- `pDevice`: D3D11 device pointer from application
- `format`: Render target resource format
- `width`: Render target width
- `height`: Render target height
- `numEyes`: The number of eyes. Default is 1 for standard desktop application. For VR applications, `numEyes` should be 2.

`TSCMAA::Resize(…)` is optionally called to resize resources in TSCMAA when the application render target is resized.

```
HRESULT TSCMAA::Resize(ID3D11Device * pDevice,
                       DXGI_FOARMAT format,
                       int width,
                       int height);
```

- `pDevice`: D3D11 device pointer from application
- `format`: Render target resource format
- `width`: Render target width
- `height`: Render target height

`TSCMAA::Draw(…)` applies TSCMAA-given color and depth resources and returns the final TSCMAA resource to `ppOutTex`. To provide input color and depth resources from the application to TSCMAA, the application should prepare `ColorDepthIn` by calling `ColorDepthIn::Create(…)`.

```
HRESULT TSCMAA::Draw(ID3D11DeviceContext * pContext,
                     ColorDepthIn * pColorDepthIn,
                     DirectX::XMFLOAT4x4 &projection,
                     DirectX::XMFLOAT4x4 &view,
                     ID3D11Texture2D ** ppOutTex,
                     unsigned int eye = 0);
```

- `pContext`: D3D11 device context pointer from the application
- `pColorDepthIn`: A pointer of `ColorDepthIn` for TSCMAA that is created in the application side by calling `ColorDepthIn::Create(…)`

- `projection`: Projection matrix
- `view`: View matrix
- `ppOutTex`: A pointer of TSCMAA output texture resource, which resides in the TSCMAA side
- `eye`: eye indexm where the left eye is 0 and the right eye is 1

To destroy TSCMAA, call `TSCMAA::Destroy()` which will also release all resources in TSCMAA.

```
void TSCMAA::Destroy();
```

TSCMAA has a few control knobs to adjust the number of edges. Since CMAA and TAA passes run only on edge candidates, the number of edges decides performance and quality. To adjust the number of edge pixels detected, TSCMAA provides `SetEdgeThresholds(…)` which will set an edge threshold and a non-dominant edge removal amount for edge detection.

```
void TSCMAABase::SetEdgeThresholds(float edgeDetectionThreshold,
                                   float nonDominantEdgeRemovalAmount,
                                   float bluriness);
```

- `edgeDetectionThreshold`: The recommended value ranges [(1.f/32.f), (1.f/1.f)] and default value is (1.f/22.f).
- `nonDominantEdgeRemovalAmount`: The recommended value ranges [0.f, 3.f] and default value is 0.5f.
- `bluriness`: The recommended value ranges [0.5f, 2.f] and default value is 0.7f.

Note that `bluriness` does not affect edge detection but does affect processing edge candidates. Hence `edgeDetectionThreshold` and `nonDominantEdgeRemovalAmount` are knobs to control the number of edges in edge candidates.

To get the current edge thresholds, use `TSCMAA::GetEdgeThresholds(…)`.

```
void TSCMAABase::GetEdgeThresholds(float &edgeDetectionThreshold,
                                   float &nonDominantEdgeRemovalAmount,
                                   float &bluriness);
```

## How to Integrate TSCMAA into Other DirectX 11* Applications

To integrate the TSCMAA library into the application:

1. Build the TSCMAA library with “Intel/TSCMAA.sln” if you do not have TSCMAA.lib.
2. Link “Intel/lib/TSCMAA.lib” to your application project.
3. Include “Intel/TSCMAA/TSCMAA.h” in your application.
4. Create a TSCMAA instance and ColorDepthIn instance.
5. Create resource views from application’s color and depth textures for TSCMAA by calling ColorDepthIn::Create(…). To create color and depth textures, use one of the supported formats in TSCMAA.
6. Initialize TSCMAA by calling TSCMAA::Create().
7. Render color and depth textures in an application render loop.
8. Apply TSCMAA by calling TSCMAA::Draw() after application render.
9. Submit TSCMAA output to HMD (or backbuffer).
10. Repeat steps 7–9 for every frame.
11. Destroy TSCMAA resources by calling TSCMAA::Destroy() and ColorDepthIn::Destroy().

The sample code is shown below.

```
#include "Intel/TSCMAA/TSCMAA.h"
TSCMAA::TSCMAA _tscmaa;
TSCMAA::ColorDepthIn _tscmaaColorDepthIn;

// In application init
_tscmaaColorDepthIn.Create(pDevice, pEyeTex, pEyeDepthTex);
_tscmaa.Create(pDevice, DXGI_FORMAT_B8G8R8A8_TYPELESS, 1080, 1200, 2);

// In app render loop
for (each frame){
    ID3D11Texture2D * pEyeTSCMAAOutTex[2] = { nullptr, };
    for (each eye) {
        // App renders into pEyeTex and pEyeDepthTex
        // …
        _tscmaa.Draw(pContext,
                     projectionMat,
                     viewMat,
                     _tscmaaColorDepthIn,
                     &pEyeTSCMAAOutTex[eye],
                     eye);
    }
    // Submit pEyeTSCMAAOutTex[2] to your HMD back buffer
    // …
}

// In application destroy
_tscmaaColorDepthIn.Destroy();
_tscmaa.Destroy();
```

## Summary

This article described TSCMAA and how easily VR application developers can integrate TSCMAA into their applications as a MSAA alternate with competitive spatial quality and better temporal stability. TSCMAA quality and performance is also adjustable with the number of edges since CMAA and TAA passes are processed only on the edge candidates, resulting in faster, less blurred pixels. The attached TSCMAA sample package has been optimized on Intel HD Graphics 630, but it runs on any platforms.