

Step by step recipe for Temporal Anti-Aliasing

example program: [google drive](#)

source code: [github](#)

Written by Ziyad Barakat

Special thanks to Xi Ma Chen for his help and patience

Table of contents

1. What is TAA?
2. A brief history
3. Ingredients
4. The high-level explanation
 - a. Initialization
 - b. Draw sequence
 - c. Preparing the next frame
5. TAA in depth
6. Other subjects
 - a. Jittering
 - b. Velocity
 - c. History buffers
 - d. Ghosting
 - e. Halton [2, 3] sequence
 - f. Framebuffer configurations
 - g. Uniform buffers
 - h. Side notes
 - i. Dithering Opacity
 - ii. Sharpening
 - iii. TXAA
 - iv. TSMAA
 - v. Quincunx shape
 - vi. Other uses
 - i. My criticisms
7. Performance
 - a. GPU and CPU time
8. Glossary

What is TAA?

Temporal Anti-Aliasing (TAA) is an Anti-Aliasing (AA) method that utilizes temporal filtering methods to improve AA stability in motion.

What differentiates it from other AA methods is that TAA produces visual results that are better or equal to other AA methods whilst being less performance intensive than other AA methods.

Not only this but TAA can also be used in other areas such as smoothing shadow maps and smoother screen space ambient occlusion techniques.

A brief history

- Before being used in real time graphics applications, TAA was first developed as a technique for films that utilize 3D graphics in the 1980s
- In 2011, a form of TAA began to be utilized in real time applications starting with Crysis 2. This implementation has an emphasis on motion blur
- Exploded in popularity when it was discussed in a 2014 SIGGRAPH presentation given by Epic Games who developed their own take on TAA using super-sampling
- Utilized in DOOM 2016 which uses a similar super-sampling method coupled with TAA
- Also used in Horizon: Zero Dawn via the decima engine which couples TAA with FXAA
- The developers of Inside also adapted TAA into the Unity Engine which they presented in a 2016 GDC talk which I followed very closely for this project.

Ingredients

- Modern OpenGL context (preferably core profile) with double buffer rendering
- Geometry buffer including:
 - Color buffer
 - Velocity buffer
 - Depth buffer
- History buffer. An array containing 2 geometry buffers (for storing the previous frame)
 - A bool/flag used for toggling which history buffer is active for a given frame
- Uniform Buffers
 - velocity reprojection
 - TAA settings
 - Jittering settings
- A textured model (preferably a highly aliased model like a tree)
- The geometry translation, camera projection and view matrices.

Shaders

Geometry/Jitter pass

[Jitter vertex shader](#)

[Velocity fragment shader](#)

TAA pass

[Default vertex shader](#)

[Smooth fragment shader](#)

Final pass

[Default vert quad](#) (or blit to backbuffer)

[Texture fragment shader](#)

Back buffer: RGBA8, 32-bit depth, double Buffered

The high-level explanation

Initialization

1. Create framebuffers
2. Create history buffer
3. Load shaders
4. Create Halton [2, 3] number values
5. Load model
6. Load textures
7. 3D camera with projection in perspective mode

The draw sequences

1. Update the uniform buffers
2. Set camera projection to perspective mode
3. Jitter pass
 - Vertex:
 - Apply jittering. Explained later
 - Use the previous and current frame camera data to create 2 sets of positional data that will be used in the next shader stage
 - Fragment:
 - determine velocity using the 2 sets of positional data and store that in the velocity buffer.
 - If necessary, apply dithering to avoid potential depth sorting issues with textures that have transparency.
4. Change camera to orthogonal projection for 2D drawing
5. TAA pass – take the velocity with current and previous depth and color render textures which are used to determine how much to blur the current and previous scene.
 - Vertex: Pass through quad shader
 - Fragment:
 - Get the UV of the closest depth value within the depth buffer of the current frame in a 3x3 kernel.
 - Use that UV data to sample from the velocity buffer which is then set in the negative. The velocity UV is then used to sample the depth of the history buffer at that location.
 - Next is the resolve function which will return the final image (too long to be left here)
6. Final Pass
 - Draw final TAA texture to back buffer / (or blit current history buffer to backbuffer)
7. Prepare to render the next frame
8. Repeat steps 1-7

Preparing the next frame

1. Swap back buffers
2. Clear frame buffers
3. Clear back buffer
4. Update the uniforms

```
velocityUniforms.data.previousProjection = sceneCamera->projection;  
velocityUniforms.data.previousView = sceneCamera->view;  
velocityUniforms.data.prevTranslation = testModel->makeTransform();  
velocityUniforms.Update();
```

5. Clear history buffer and swap the active buffer for the next draw. (flip the history buffer bool)

TAA in depth

TAA pass

This is the final phase of TAA where the depth, color or luma(light) of both the current and previous scene as well as velocity are used to determine when to and how much to blend the current scene with the previous scene.

Blending with the current scene too much will not make much of a difference especially in motion whereas leaning too much on the previous scene will cause ghosting artifacts (this is essentially pseudo recursive rendering).

My implementation simply gets the average depth of pixel in a 3x3 neighborhood and checks whether that average is higher than an arbitrary value. If the average is too low, then we can guess that there is nothing there at that pixel or its very edge or an object, so we stop blending with the previous scene to cut down on ghosting artifacts.

TAA resolve functions

There are a number of resolve functions that we can use to boost the visual quality of TAA which currently exist but for this project I picked the Inside TAA resolve method particularly because of the visual results in addition to the fact that this resolve method is much easier to understand than others I've found.

For brevity [here](#) is a link to the original paper written by the developers of Inside

Jittering

Jittering is an AA method typically used for static scenes which works by very subtly moving (or shaking) world geometry in (viewspace?), by using an array of evenly distributed noise values (Halton [2, 3] sequence). The amount of movement cannot exceed that of 1 pixel or else the jittering will cross over from sub-pixel blurring (creating a nice AA effect on all affected geometry) to full on blurring. The jittered pixel is then naturally blended over multiple frames (TAA pass) to smooth jagged edges.

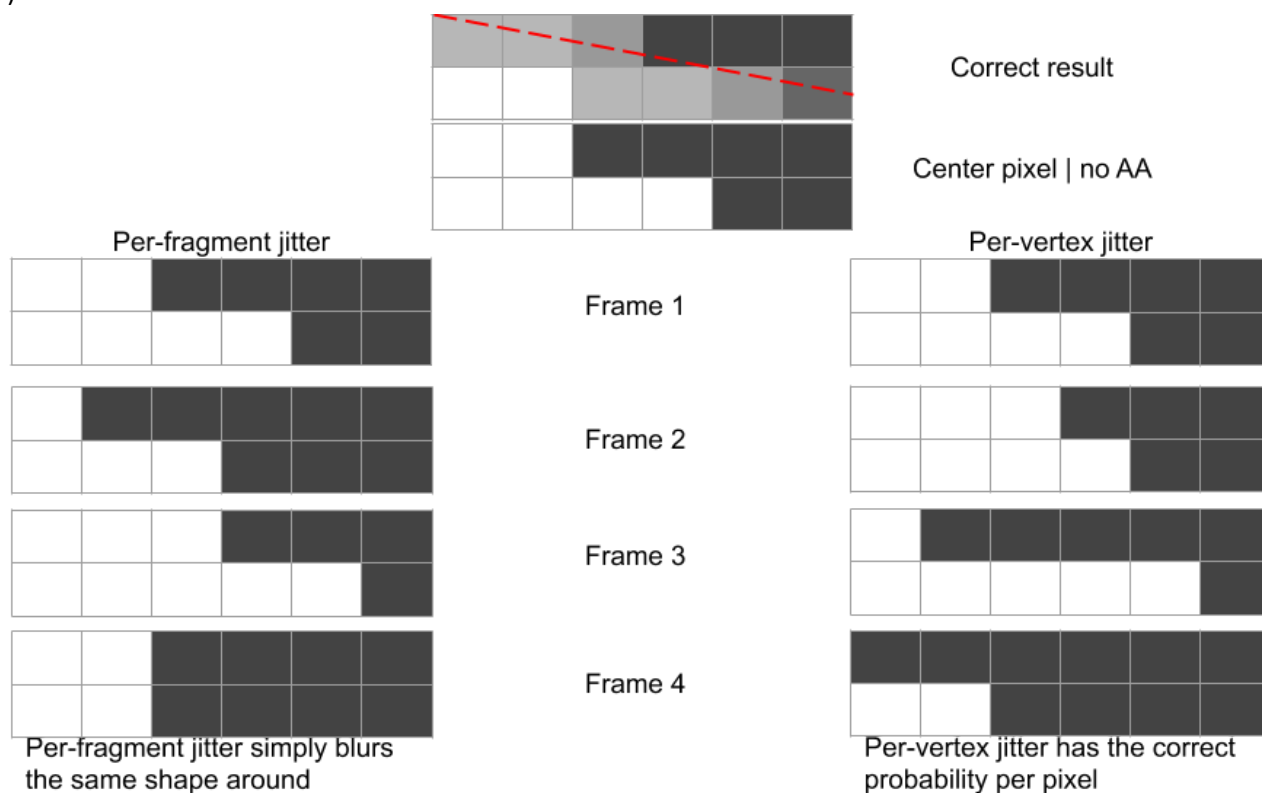
Jittering is done via a vertex shader (jitter.vert) by manipulating the projection matrix like below for AA in a static scene. For determining velocity, it's important to use the unjittered current scene and previous scene so it's a better idea to make a copy of the projection and manipulate that.

```
float deltaWidth = 1.0 / resolution.x;
float deltaHeight = 1.0 / resolution.y;

uint index = totalFrames % numSamples;
vec2 jitter = vec2(haltonSequence[index].x * deltaWidth, haltonSequence[index].y *
deltaHeight);

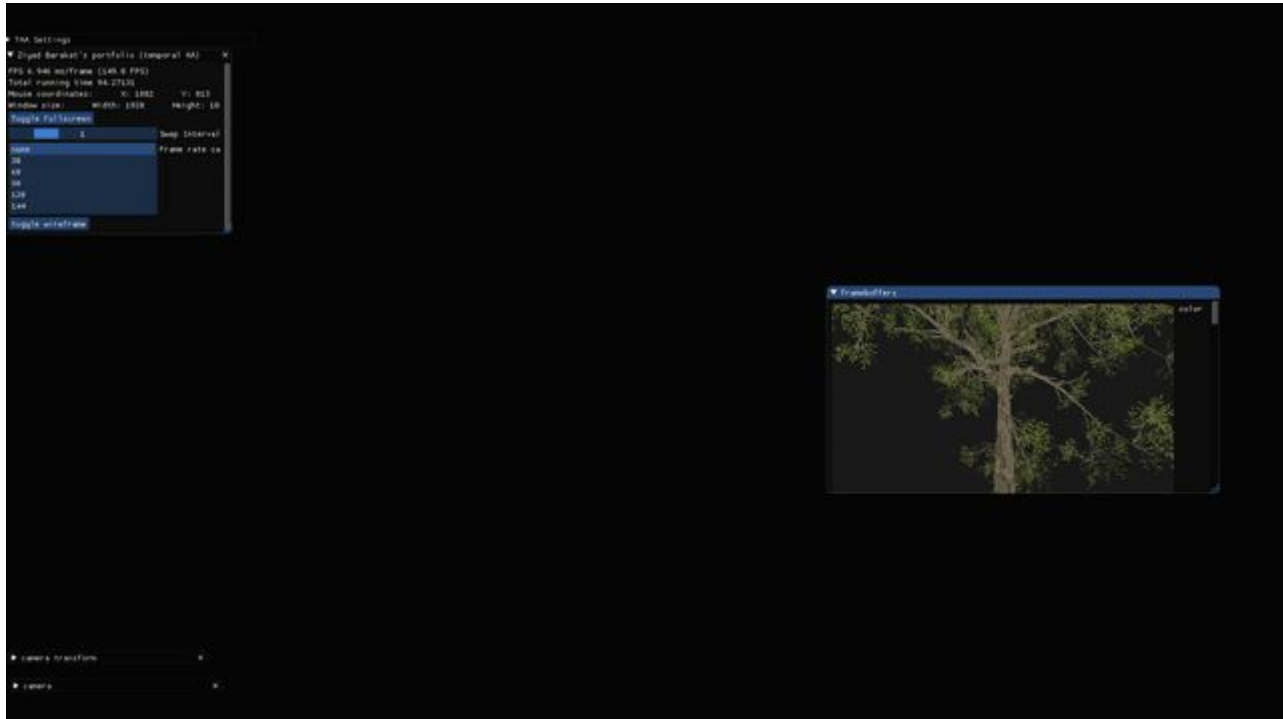
mat4 newProj = projection;
if(haltonScale > 0)
{
    newProj[3][0] += jitter.x * haltonScale;
    newProj[3][1] += jitter.y * haltonScale;
}
```

Below is an image that illustrates why jittering the vertices instead of jittering every pixel of an image is preferable. Also be sure to not apply jittering in the fragment shader as this will result in an incorrect blur. (as well as blur the entire image instead of the geometry, ignoring depth based blur so things further away appear blurrier than they should).



Velocity

We use the previous camera View and geometry translation matrices in the jitter vertex shader so can have both the current and previous jittered positions of the geometry. This data is used in the next step of the shader pipeline in order to generate the velocity information needed by calculating the positional (in screen space) difference between the geometry in the current and previous frames.



This velocity value will then be stored in a render texture to be later used when applying the final TAA effect. When used in the TAA pass, it is needed for determining where to sample from the history depth buffer. The larger the depth value (in either positive or negative values), the further away the previous depth is sampled as a parameter in the final blending for TAA.

Calculating velocity – vertex shader

During the geometry pass get the difference in position from the current frame and the previous frame. First create 2 extra vertex shader outputs and render the current and previous scene to these outputs by using the view and translation matrices of the current and previous scene.

```
//these 2 should be in screen space
outBlock.newPos = projection * view * translation * position;
outBlock.prePos = projection * previousView * previousTranslation * position;
```

Be sure to save the view and translation matrices from the previous frame to use here. Also note that when calculating velocity, you do not want to use the jittered positions which can cause the velocity to be inaccurate.

Calculating velocity – fragment shader

First move the outputs from the vertex shader from screen space (-1 to 1) to UV space (0 to 1).

```
vec2 newPos = ((inBlock.newPos.xy / inBlock.newPos.w) * 0.5 + 0.5);
vec2 prePos = ((inBlock.prePos.xy / inBlock.prePos.w) * 0.5 + 0.5);
```

Then calculate velocity as (newPos – prePos) and output that to the velocity render texture

NOTE: velocity values can be incredibly small which would require the use of a high precision texture (which can take a lot of memory). A way to get around that is to first multiply the velocity by a large number and in the next pass when reading from the velocity texture, divide the value by that number to restore the original values. This is done for those values to be saved to a lower precision texture (0.01 -> 1(lower precision))

History Buffers

These are needed for blending between the current and previous frames using a current frame flag (usually a bool) for determining the active history to draw from, draw to and clear buffers. This flag will be flipped between the 2 states every frame to swap which buffer does what every frame.

During the TAA pass, the selected history buffer will have its colour and depth used for blending the current and previous frames together. After rendering, remember to first clear the previous framebuffer and copy the current depth from the geometry buffer into the depth attachment of the current history buffer (before flipping the current frame flag).

```
historyFrames[!currentFrame]->Bind(); //clear the previous, the next frame current
becomes previous
historyFrames[!currentFrame]->ClearTexture(historyFrames[!currentFrame]->attachments
[0], clearColor1);
historyFrames[!currentFrame]->ClearTexture(historyFrames[!currentFrame]->attachments
[1], clearColor2);
//copy current depth to previous or vice versa?
historyFrames[currentFrame]->attachments[1]->Copy(geometryBuffer->attachments[2]);
//copy depth over
```

Ghosting

Ghosting is caused by a combination of when the current and previous frames are mixed improperly, resulting in accumulated data that is carried over from one draw call frame to the next, creating a "wispy" effect when the camera or scene is moving at a rapid speed.

This occurs during the final shader pass which blends both the current and previous frames together (depending on pixel velocity), to smooth the image whilst in motion. However, there are ways to mitigate ghosting with the most basic solution by doing a neighbourhood depth test (of the current frame) in the final TAA shader.

```
float averageDepth = 0;
for(uint iter = 0; iter < kNeighborsCount; iter++)
{
    averageDepth += curNeighborDepths[iter];
}

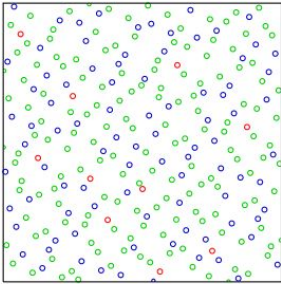
averageDepth /= kNeighborsCount;

//for dithered edges, detect if the edge has been dithered?
//use a 3x3 grid to see if anything around it has high enough depth?
if(averageDepth < maxDepthFalloff)
{
    res = taa;
}

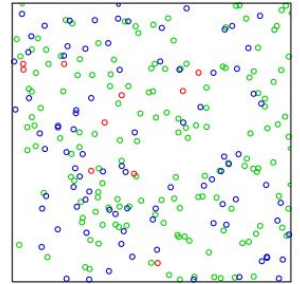
else
{
    res = texture2D(currentColorTex, inBlock.uv);;
}
```

Halton [2, 3] sequence

When Jittering, a two-dimensional array of noise is used to move the scene around in an even spread. To get this even spread, the Halton 2, 3 sequence is preferable as this sequence produces a range of random numbers evenly covering a wide range that is much smoother than other noise generating sequences.



https://en.wikipedia.org/wiki/Halton_sequence



Creating the sequence:

```
glm::vec2 haltonSequence[128];
float CreateHaltonSequence(unsigned int index, int base)
{
    float f = 1;
    float r = 0;
    int current = index;
    do
    {
        f = f / base;
        r = r + f * (current % base);
        current = glm::floor(current / base);
    } while (current > 0);

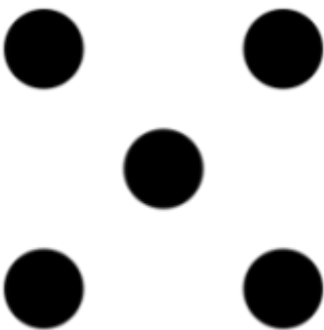
    return r;
}
```

On start-up:

```
for (int iter = 0; iter < 128; iter++)
{
    jitterUniforms.data.haltonSequence[iter] = glm::vec2(CreateHaltonSequence(iter + 1, 2), CreateHaltonSequence(iter + 1, 3));
}
```

Quincunx shape

A simpler alternative to using a halton[2, 3] sequence. Essentially a 5-point star shape.



[This Photo](#) by Unknown Author
is licensed under [CC BY-SA](#)

Framebuffer configurations

resolution = ~1920 x 1080

Color buffer

Target	Data type	Format	Internal format	Min/mag filter	Wrap	Resolution	Memory
2D	Unsigned byte	RGBA	RGBA8	Linear	Repeat	Match backbuffer	4 bytes x resolution ~8.3MBytes

Velocity buffer

Target	Data type	Format	Internal format	Min/mag filter	Wrap	Resolution	Memory
2D	float	RG	RG16 signed	Linear	Repeat	Match backbuffer	(16 * 2) bytes x resolution ~4.1MBytes

Depth buffer

Target	Data type	Format	Internal format	Min/mag filter	Wrap	Resolution	Memory
2D	float	Depth component	Depth 24	Linear	Repeat	Match backbuffer	24 bytes x resolution ~6.2MBytes

History Color buffer * 2

Target	Data type	Format	Internal format	Min/mag filter	Wrap	Resolution	Memory
2D	Unsigned byte	RGBA	RGBA8	Linear	Repeat	Match backbuffer	4 bytes x resolution ~8.3MBytes

History Depth buffer * 2

Target	Data type	Format	Internal format	Min/mag filter	Wrap	Resolution	Memory
2D	float	Depth component	Depth 24	Linear	Repeat	Match backbuffer	24 bytes x resolution ~6.2MBytes

Uniform buffers

```
struct jitterSettings_t
{
    glm::vec2          haltonSequence[128];
    float             haltonScale;
    int               haltonIndex;
    int               enableDithering;
    float             ditheringScale;

    jitterSettings_t()
    {
        haltonIndex = 16;
        enableDithering = 1;
        haltonScale = 1.0f;
        ditheringScale = 0.0f;
    }
};

struct reprojectSettings_t
{
    glm::mat4          previousProjection;
    glm::mat4          previousView;
    glm::mat4          prevTranslation;

    glm::mat4          currentView;

    reprojectSettings_t()
    {
        this->previousProjection = glm::mat4(1.0f);
        this->previousView = glm::mat4(1.0f);
        this->prevTranslation = glm::mat4(1.0f);
        this->currentView = glm::mat4(1.0f);
    }
};

struct TAASettings_t
{
    //velocity
    float velocityScale;
    //Inside
    float feedbackFactor;
    //Custom
    float maxDepthFalloff;

    TAASettings_t()
    {
        this->feedbackFactor = 0.9f;
        this->maxDepthFalloff = 1.0f;
    }
};
```

```

        this->velocityScale = 1.0f;
    }
};

```

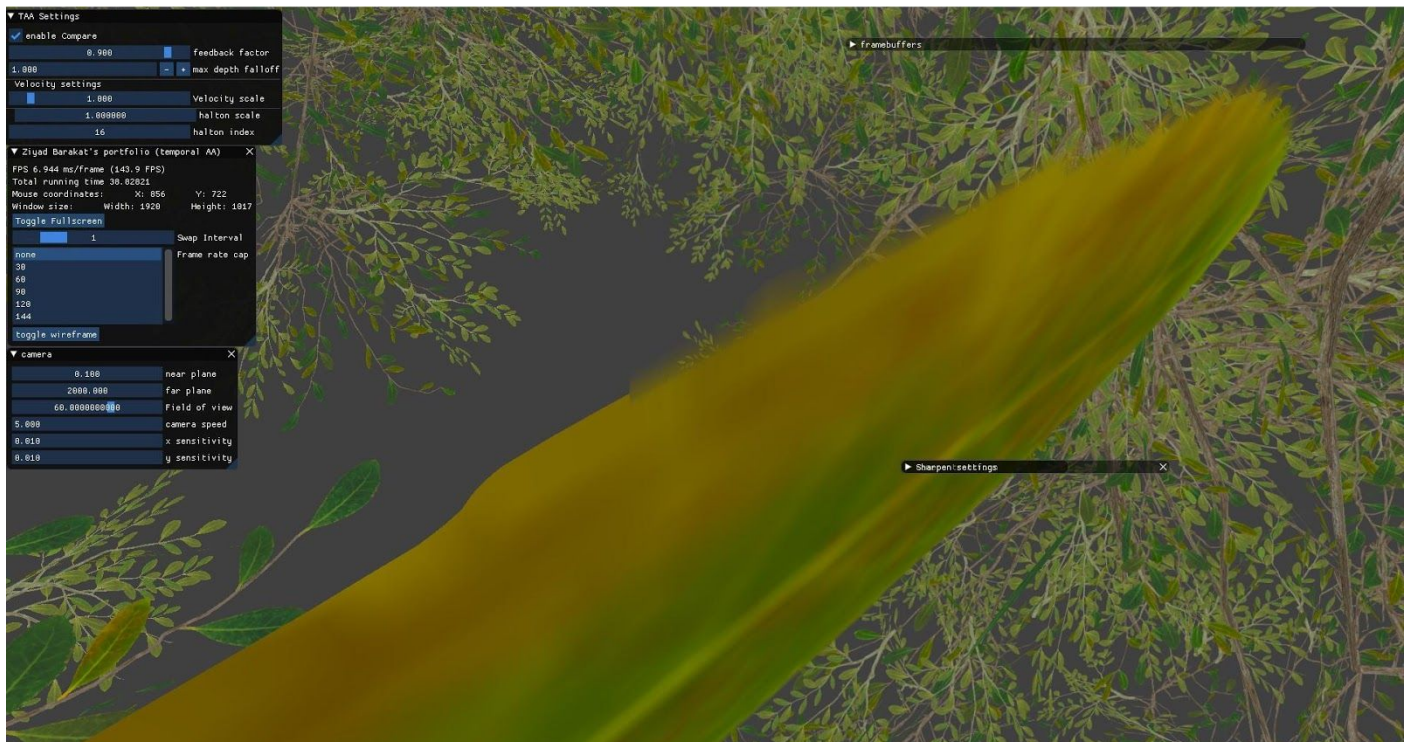
Other notes

Dithering opacity

Dithering can be combined with TAA blending to mitigate some transparency and depth related issues and can also be used to assist AA in a static scene by introducing a probability of a pixel being rendered in the geometry pass.

This probability is based on the alpha level of the pixel with the higher alphas being more likely to be drawn and lower alpha being less likely. E.g if the alpha is 0.5 then only half of those pixels should be drawn, if 0.25 then %25 should be drawn, etc.

Dithering is then smoothed out in the TAA pass by blending the current and previous scenes which will cut down the amount of visual noise usually associated with dithering.



Sharpening

You can also apply a sharpening after the TAA pass if your implementation leaves the image looking a little too “soft”, which is a common complaint about the TAA implementation in some games.

TXAA = TAA + FXAA

This is fairly straightforward, in addition to using jittering as the means of AA, an FXAA shader is used after the jitter/geometry pass (but before the TAA pass). This is a popular method of additional AA as FXAA excels at removing jagged edges from lines and grid shapes whereas jittering typically is not.

TSMAA = TAA + SMAA

This is combining Sub-pixel morphological Anti-Aliasing with temporal filtering to generate an amazingly clean final image.

Other uses

Temporal filtering can also be used in other areas of rendering such as smoothing shadow maps and ambient occlusion noise.

My criticisms

The process of learning how to implement TAA was very difficult considering a severe lack of step by step tutorials and documentation as well as a lot of the documentation that does exist is really difficult to understand for people who don't have a strong background in mathematics.

Also, both the Inside paper and the Unreal Engine 4 paper have typos with the UE4 paper having a larger typo that can easily cause issues.

Adjust projection matrix

```
ProjMatrix[2][0] += ( SampleX * 2.0f - 1.0f ) / ViewRect.Width();  
ProjMatrix[2][1] += ( SampleY * 2.0f - 1.0f ) / ViewRect.Height();
```

This is supposed to be [3][0], [3][1].


Performance

CPU and GPU Time (recorded by the [MicroProfiler library](#))

										8 min ago
Group	Timer	Average	Max		Total	Min		Call	Average	Count
CPU			6.78		6.88		406.61			
	Render		6.78		6.88		406.61		6.65	6.78 60
	Jitter pass		0.04		0.06		2.44		0.03	0.04 60
	Unjittered pass		0.02		0.03		1.11		0.01	0.02 60
	Temporal AA pass		0.02		0.04		1.02		0.01	0.02 60
	Final pass		0.01		0.01		0.32		0.00	0.01 60
	building ImGui		0.06		0.13		3.60		0.05	0.06 60
GPU			4.07		4.56		244.17			
	Jitter pass		2.00		2.49		119.71		1.94	2.00 60
	Unjittered pass		1.88		1.92		113.05		1.86	1.88 60
	Temporal AA pass		0.15		0.15		8.77		0.14	0.15 60
	Final pass		0.04		0.05		2.63		0.04	0.04 60
MAIN		0.00	0.00	0.00	0.01	0.00				
	CPU	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	60
MicroProfile		0.01	0.01	0.03	0.71					
	MicroProfileFlip	0.01	0.01	0.03	0.71	0.01	0.01	0.01	0.01	60
	ThreadLoop	0.00	0.00	0.00	0.09	0.00	0.00	0.00	0.00	60
	Clear	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	60
	Accumulate	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	60
	ContextSwitchSearch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0
	WebServerUpdate	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0

According to the above data the amount of time each render pass can take up to 2.19ms on average to render the Jitter pass, the Temporal AA pass and the final pass(draw to backbuffer) on the GPU. Also, due to the miniscule draw call amount, the amount of CPU time for each draw pass is negligible. This means that adding TAA to your project could add an additional (Jitter = (2 - 1.88 = 0.12) + TAA = 0.15) ~0.27 MS to your render time per frame.

renderDoc gives a similar story:

EID	Name	Duration (ms)
	▼ Frame #3420	4.72266
0	Capture Start	
3-5	➤ Copy/Clear Pass #1	0.01741
9-11	➤ Copy/Clear Pass #2	0.00717
15	glClearBufferfv(GL_COLOR, 0, 0.250000, 0.250000, 0.250000...	0.00512
20	glClear(Color = <0.250000, 0.250000, 0.250000, 1.000000>...	0.00819
40-56	➤ Colour Pass #1 (2 Targets + Depth)	2.03674
68-84	➤ Colour Pass #2 (1 Targets + Depth)	2.3593
106	glDrawArrays(6)	0.18227
115-202	➤  Colour Pass #3 (1 Targets + Depth)	0.10646
215	SwapBuffers(Backbuffer Color)	0.00

According to this dataset, adding jitter and velocity calculation adds roughly ~0.3ms to your geometry pass, and the TAA resolve pass can add up to 0.18ms to render time as well.

NOTE this program renders a single tree object, consisting of 3 meshes, thus only taking up 3 draw calls.