# Investigation of Temporal Reprojection Anti-Aliasing

Callum Alex Randall

*School of Computing Science, Newcastle University, UK*

**Abstract**

Temporal anti-aliasing is a form of anti-aliasing that combines information across multiple frames in order to reduce jagged edge artifacts. Unless horizontal or vertical, lines cannot be perfectly replicated by pixels and create the jagged effect when stepping across the threshold from one pixel to another. These are especially apparent when the resolution of the image is not sufficient to approximate the line. In this project, temporal anti-aliasing is investigated and implemented including projection matrix jittering, reprojection using motion vectors and history validation with neighbourhood clipping. The resulting solution successfully eliminates jagged edges with minimal artifacts as a result of motion such as ghosting. Performance is measured and has minimal impact although it becomes more expensive with higher resolutions.

*Keywords:* Anti-aliasing, history buffer, reprojection, projection jittering, neighbourhood clipping.

## 1 Introduction

Anti-aliasing techniques are required to counteract an artifact found in rasterised images called "jaggies" or "jagged edges" that occur commonly in computer graphics. These are stair like lines that appear when there should be a straight line or curve. This is due to the display resolution of most monitors and televisions not being high enough to be able to fully approximate these lines. If a diagonal line is represented on a screen, it will step across either horizontally or vertically as it crosses the threshold from one pixel to another creating a stair like line that is not smooth and is jagged.

Figure 1 shows how a triangle is rendered to screen without anti-aliasing. The black dotted line represents the true edge of the triangle and in the image it can be seen that the black dotted line crosses over some pixels that are not shaded blue, even though the triangle edge crosses through this region and should be visually represented in some way. This is because a pixel is only rendered if the geometry of the shape crosses either the center of the pixel or covers more than half of the pixel area, depending on the engine. Since the black dotted line only cuts across

[1] Email: c.a.randall@ncl.ac.uk

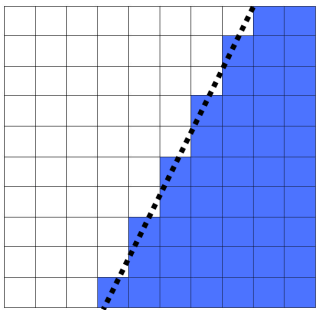

Fig. 1. Zoomed in approximation of a triangle edge

the corner, nothing will be visually represented and instead the stair like jagged line will occur.

A way to solve this would be to increase the resolution of the rendered image which would make the density of pixels higher and thus the artifacts would become less noticeable. Increasing the resolution is less feasible in video games as they run in real time with typically 60 frames or upward being rendered and displayed per second, rasterising more pixels per frame increases the workload of a GPU (graphics processing unit) increasing the time it takes to render each frame. An anti-aliasing solution does not require increasing the resolution to reduce the artifacts and results in a smoother image with less time required for the GPU to render each frame.

There are various different ways anti-aliasing can be implemented. The primary benefit of TAA (temporal anti-aliasing) which is the subject of this report is that it is much faster than other methods such as MSAA (multisampling anti-aliasing) which takes a sample of a pixel multiple times per frame at different locations and takes an average of these samples to produce a final pixel. TAA samples each pixel only once however it does it multiple times over different frames and uses a history buffer to accumulate pixels from several frames to create a smoother image from previous information. This is faster than MSAA because it takes place over several frames rather than calculated every frame and achieves similar quality.

### *1.1 Aim and objectives*

The aim of this project is to develop a temporal reprojection anti-aliasing solution and investigate the performance characteristics and visual quality improvements of the solution.

- Understand how projection matrix jittering, reprojection and history rectification can be used to create a temporal anti-aliasing solution
- Investigate which engine or codebase is most suitable to implement the solution for this project.
- Implement projection matrix jittering with a suitable distribution pattern.
- Implement a resolve pass with a history buffer which accumulates multiple frames and displays an anti-aliased image for static scenes.
- Implement reprojection using motion vectors to correctly sample from the history buffer in the correct location for dynamic scenes.
- Implement neighbourhood clipping to rectify invalid history samples and reduce

visual artifacts.

- Analyse the visual qualities of the solution in multiple scenarios and collect data on the performance impact.

# 2 Background

## *2.1 Overview*

An overview of the implementation of the temporal anti-aliasing technique including which steps need to be applied in which order and how they relate to each other should be thoroughly explored before beginning the implementation of the method itself. A full understanding of the process and how each part relates to the next is required to be able to implement the technique correctly.

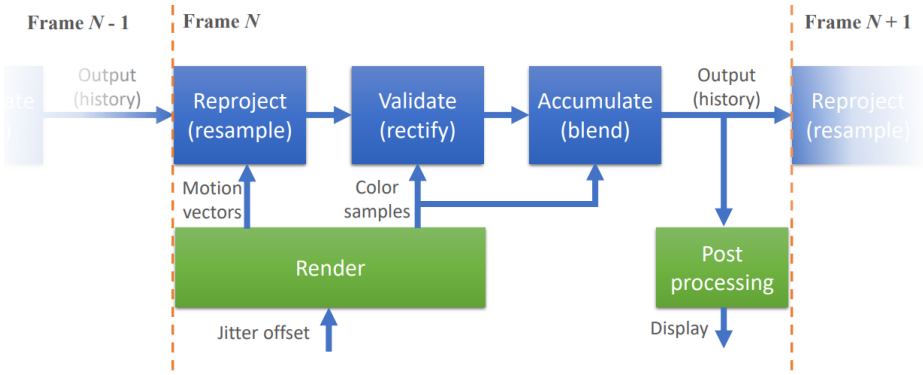


Fig. 2. Temporal anti-aliasing implementation diagram

Figure 2 shows a common implementation of a temporal anti-aliasing solution from Yang et al. [1] and represents a simple overview of the steps that are taken during each stage of the technique. The general steps of how to implement a temporal anti-aliasing solution into a game engine are as follows.

A jitter offset is applied to the projection matrix of the in-game camera each frame. The offset is usually determined by a quasi-random sequence in order to evenly sample different locations.

Samples from the previous frame output are temporally accumulated and are then reprojected using motion vectors to account for motion of the camera and objects in the scene.

Each frame then must be validated as history samples can become invalid due to occlusion and disocclusion of various objects in the game scene. The history sample therefore must be constrained in some fashion in order to only sample valid data.

The constrained history sample is then accumulated with the new sample to obtain a resolved colour for each pixel. This can be weighted in various ways in order to obtain different results. The frame that is to be rendered is then saved within the history buffer to be used in future passes.

Each stage of this process will be expanded on to provide further understanding of how they function.

### *2.2 Jittering*

Temporal anti-aliasing was popularised with games such as Crysis 1 and 2, Killzone 2 and Halo 3 [2] and was initially introduced with a focus on motion blur by applying a directional blur from the previous frame to the current frame using a velocity vector. This technique was combined with the OpenGL Accumulation Buffer SSAA [2] which added sub-pixel jittering of the camera view frustum with a combination of sub-samples.

As was established earlier, the advantage of a temporal anti-aliasing technique is that only one sample per pixel is required to be rendered during each frame. This is advantageous in the respect that each pixel being sampled only once means that for each frame the GPU only needs to render the pixel in one location rather than in MSAA where it is sampled multiple times in different locations per frame.

Temporal accumulation works well when there are small amounts of either camera motion or object motion as it can sample the history of the pixel and blend it accordingly, but problems arise when the scene is static. If the camera and game objects do not move then information in the history buffer is not being gained as it will sample the same pixel in the same place each frame and no new information is returned. This is the advantage of jittering in a temporal solution as even in a static scene the camera is moving small amounts each frame. Information about each individual pixel is being accumulated and blended into each frame so that even if the camera is static, information about each pixel is being gained from the small jitter applied to the camera view frustum.

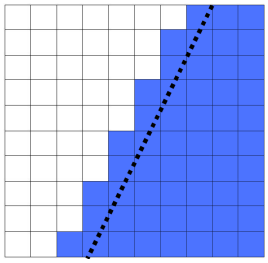

Fig. 3. Figure 1 shifted to the left by a pixel

Figure 3 shows the same triangle edge diagram as shown in Figure 1 but it has been shifted by a pixel to the left. Although jittering tends to happen on a sub-pixel basis this is just an example to show what jittering achieves. The black dotted line that shows the true triangle edge used to cross through some pixels but did not cross the center of the pixel region, causing some pixels to not be visually represented in the final render. Now that the projection matrix has been shifted a small amount, the pixels that should have been represented previously are now shaded. This information will then be blended into the history buffer. It then jitters in various different directions each frame and accumulates the missing details that had been missed from just a static render and blends them with the history buffer. Even though the pixel may have only been shaded blue for just one frame, that small amount will be blended into the history buffer so that the edges that do not cross the center of the pixel will have some influence on the final colour of the pixel.

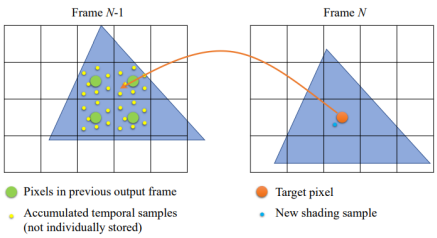


Fig. 4. Accumulation of samples over multiple frames [1]

Figure 4 shows an illustration of how subpixel samples over multiple frames can achieve supersampling and provide more detail. The yellow dots represent the multiple samples that have been accumulated over multiple frames and blended into the green dot which is stored in the history buffer. The orange dot shows that when the object moves, it samples the history buffer from the previous location to get the history colour for that pixel. This history colour represents an average of accumulated samples around that point, providing more detail over multiple frames.

A major consideration to make when implementing jittering into the solution is the sample distribution that will be used to allow for the best speed of convergence and quality. The speed of convergence is the speed in which the image will resolve itself. One of the first implementations from Sousa [3] used a 2x Quincunx distribution that had two sub-samples however a more suitable solution that could be used for this project can be found within developer Playdead's temporal anti-aliasing solution developed for the game INSIDE [4].

Karis [5] found that the Halton sequence (2,3) was a suitable sample pattern that avoided clustering in space or time. Playdead used the first sixteen samples of this sequence to create their sample distribution for their implementation in INSIDE.

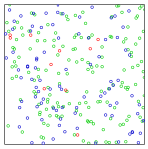

Fig. 5. A pseudorandom distribution [6]

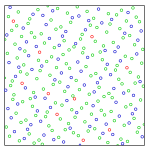

Fig. 6. 256 samples of Halton(2,3) sequence. [6]

Figures 5 and 6 shows the difference between a generic pseudorandom distribution and the quasi-random nature of the Halton sequence. The Halton sequence more evenly covers the space whilst also maintaining a random appearance. This enables the jittering of the camera to be distributed evenly and cover all angles which will allow for an accurate resolve whilst also maintaining a random quality that will allow the jittering to be suitable in all scenarios.

After jittering the camera with this sample distribution, a resolve pass would allow a static scene to render without jagged edges by combining the sampled colours from the small jittering applied to the camera and blending them into a single colour. The resolve pass is done by blending the current frame with the history buffer by a certain weight factor and then rendering the result of the blend to screen. The result is also passed into the history buffer and will be used in the next resolve pass.

### *2.3 Reprojection*

In a dynamic scene, objects are moving across the screen and the camera itself may pan across the scene. This results in objects moving across the frame which becomes problematic when sampling pixel data. If the temporal pass is sampling the pixel in the exact same location each frame yet the object itself is moving then the history data becomes invalid as the object has moved. For each pixel, the corresponding location in the previous frame must be found in order to fetch history data correctly. If this is not done, then it will result in a ghosting effect. A trailing artifact will appear on the screen as the temporal system is sampling the previous frame at the same position as the current frame, but since the object has moved in the screen space the sampling is now incorrect.

Fig. 7. An example of ghosting. [7]

This can be achieved in a few ways, one of which is reverse reprojection. As explained in the paper [8] this is when geometry is transformed during the rasterisation of the scene twice. Once is done using the data from the previous frame and another using the current frame. The current location of the pixel and the previous position of the pixel is used to create an offset that is stored into typically a motion vector texture or just a velocity in some game engines[1]. This motion vector texture stores the per pixel screen space motion of objects on the screen from one frame to the next. This velocity can be used to reconstruct previous positions in the history buffer texture. Locations of pixels change as they move on the screen and using this motion vector texture we can now sample the pixel from the history texture in the correct position as we know the offset between its previous and current position. This means that even if an object is moving the pixel shader can sample the object in the correct position. The ghosting effect in Figure 7 will now now longer occur when the camera is changing angle however when the camera or the object moves there may still be ghosting trail that cannot be fixed through reprojection alone.

Another method of reprojection is possible and is necessary in some game engines

as the motion vector texture that was used in the previous method is sometimes not calculated to save framebuffer bandwith[1]. An alternative of this is to use the current and previous frame view and projection matrices along with the world matrix for the objects in the scene.

In the vertex shader this can be achieved by taking the position of the vertex and multiplying the matrix with the world matrix of the current object. The result of this multiplication will then be multiplied with the view projection matrix in order to translate the position of the vertex into clip space [9]. This is the stage where it is determined which portions of an object will be visible and which will be "clipped". Finally it will be converted from clip space into NDC (normalised device coordinates) space.

Doing this for both the current position vertex and the previous position vertex will result in the NDC position of the object from the current frame and the previous frame. Subtracting the previous NDC position from the current will result in an offset that represents the velocity. The offset represents the change in position from one frame to the next. It can be used to reproject just like the motion vector texture earlier by subtracting this value from the sample position of the history texture coordinate in the pixel shader. This will sample the colour in the history texture of the previous position so that the correct sample can be used to blend with the current sample. This can also be achieved by using the depth buffer from the camera and projection matrices to reproject onto the previous camera position screen space and sample the history texture from there.

### *2.4 History Rectification*

Reprojected data from the previous frame should not always be reused trivially without further checking the validity of the sample. There are various ways that a sample of the previous frame may be invalid such as occlusion and disocclusion. An example of this would be if an object moves in front of another, a sample of the previous position of the occluded object may sample the new object and would not be correct. Differences in lighting and shading may also cause this, a light source changes colour or position and will cause the sample to be invalid as the previous frame was different to the current one. Failure to properly sample valid history samples will result in more artifacts such as the ghosting shown in Figure 7.

A technique not used so much now but was used in the earlier stages of temporal anti-aliasing development was history rejection. This would simply detect if an error has happened when sampling the previous data and would set the blending factor to either zero or one, effectively clearing the history sample and passing only data from the current frame into the buffer. Soft refresh of the history buffer is also possible by limiting the blending factor by a threshold when an error is detected so that the buffer isn't completely lost and accumulation of history data can happen faster, resulting in less artifacts [1]. The data that was used as the source to detect if an error has happened was either data from the geometry such as depths, normals and motion vectors or just colour data. Geometry data can be used to detect occlusion and reject the history data if an occlusion has been found. Sousa [3] had used an approach like this in CryENGINE 3.

History rectification improves upon simply rejecting erroneous history samples, the main drawback of that method being that the history data buffer is reset once an error is found which can lead to artifacts that show before the history buffer accumulates again. Rectification instead aims to use the rejected data but makes the data more congruent with the samples from the current frame. The history samples that have been rectified are then blended into the current frame and leads to less artifacting and more consistent visuals.

Colour clamping is a simple and effective solution to determine which colour samples can be used in the accumulation process. An approach described in [10] by Malan is to take the history pixel and compare it to the current pixel neighbourhood. The neighbourhood of a pixel is the pixel itself and surrounding pixels in an predefined area. An assumption is made that surrounding pixels provide a better estimation of colour variations in a local area. Sousa further developed this in the CryENGINE [11] by taking the range of colours that are found within the neighbourhood as the minimum and maximum extents of a clamp. This means that if the history colour falls within the extents of the neighbourhood it is passed through as it is, but if it is beyond the extents it is clamped so that the sample colour will only be between the minimum and maximum extent. By reusing the history sample when it is incorrect by just clamping the colour that is passed it sacrifices some accuracy of what is being sampled however it will reduce artifacts such as ghosting as clear reprojection errors will be rectified and clamped to the local neighbourhood values.

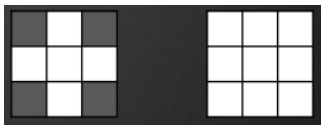

Fig. 8. Neighbourhoods used by Karis. [5]

Figure 8 shows the two different neighbourhoods that are used within the implementation by Karis[5]. Initially Karis used only a 3x3 neighbourhood but there were artifacts due to the shape. He then rounded out the shape by adding the plus shaped neighbourhood and averaging the two seperate neighbourhoods to get a filtered appearance. This is a more expensive approach on the GPU as more neighbours need to be sampled however it leads to an increase in image quality.

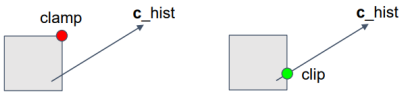


Fig. 9. Clipping example from Playdead [4]

Figure 9 shows an approach to even further improve this method of history rectification by instead clipping rather than clamping the current sample. As shown by the image, clamping can result in clustering when a sample is very distant from the history sample. It could result in convergence towards the so called corners of the colour extents so that rectified colour samples converge towards a single colour whereas clipping would allow a more realistic and less clustered variety of colours.

# 3 Design

### *3.1 Software Choice*

After research of temporal anti-aliasing solutions had concluded, the design of the implementation of the technique could begin. It is necessary to choose the correct environment to implement the technique with respects to the project itself. A game engine or codebase must be suitable to properly implement the technique fully but since there are multiple engines that could be chosen they should be compared to see which advantages each one has compared to the other and make a decision.

Two of the largest current game engines that are popular to use for personal projects and general development are Unity Engine and Unreal Engine. Choosing between one of these can come down to personal preference and experience as they are both quality engines for the purpose of this project. A simple search for documentation relating to applicable topics within the temporal anti-aliasing solution such as motion vectors and general shader documentation provided more confidence to move forward with Unity as the engine of choice out of the two largest general purpose game engines. Previous personal experience with the engine also helped consolidate the choice as less time would need to be spent learning how to use the engine at a surface level. Learning would be required with both options to implement the temporal solution itself but basic operation of the engine was already known.

Another option that could have been chosen for this project is to use a framework that had been used throughout the Game Engineering course at Newcastle University. The framework consists of a Visual Studio solution with C++ and OpenGL integration. The framework provides an environment in which essential data to complete the implementation such as the projection and view matrix was all available and could easily have been chosen to complete the project. Ultimately it was not chosen to be used for the project for a few reasons.

Simple things such as just being able to easily use a slider to be able to zoom in the game view image in the Unity engine would prove helpful with inspecting further away details which are heavily effected by temporal anti-aliasing. Unity also easily allows switching between scenes to be able to evaluate the temporal effect in various different scenarios which would make the evaluation stage of the project easier to complete. A large reason for opting to not use the OpenGL framework is it takes a lot more work to be able to import a game scene which can be used to test the implementation easily. Unity has various different means such as the Unity Asset Store that allows for importing of objects and environments which allows the implementation itself to be tested and time would not be wasted either creating a game scene to test from scratch or further expanding the framework with more code to be able to handle importing and handling larger game scenes that are more typical in an actual video game. This allows for testing in an environment which is more representative of the one the solution would be applied to.

### *3.2 Solution Design*

Next was to move forward with the implementation of the solution itself. Since the jittering needs to applied to the camera object in the scene, it made sense to approach the solution by creating a script that can be attached to a camera object in Unity. This allows for the temporal anti-aliasing to be easily used in various different projects as it only requires a script and a shader attached to a camera object which can easily be imported into any Unity project. Being able to import the solution into various different scenes and projects would make it easier to be able to test the technique further down the line as various different scenarios can be used to test multiple different problematic areas that arise with temporal solutions to test how this implementation handles these problems.

A few different scenarios were chosen to test the technique. A very basic level with basic geometry was used initially especially when testing a primarily static scene just to see what the impact of the code would be during the implementation before it was fully complete and to be able to test it out on the fly. As the code progressed and was able to handle moving objects, a script was made to simply rotate the objects in the level to see the impact that motion brings to the temporal effect. The camera was also equipped with a script that allowed for panning and movement of the camera through mouse and keyboard controls. This allowed the camera to easily be moved to different angles and also analysing the effect that camera movement had on the temporal technique.

Fully designed game environments were then imported with the Unity Asset Store to be able to asses how the temporal implementation handled a full game environment that would be the typical use case of the solution. This included various different variables into the scene such as distant geometry, transparency, vegetation and flickering which would allow for interesting test cases to be evaluated in later sections of the project.

# 4 Implementation

## *4.1 Jitter implementation*

To begin the implementation of the temporal anti-aliasing solution was to create a very basic scene with simple geometry. Just cubes that are rotated slightly would be enough to be able to test in real time if the implementation was working to a minimal extent and a more complex environment can be introduced later to develop later changes.

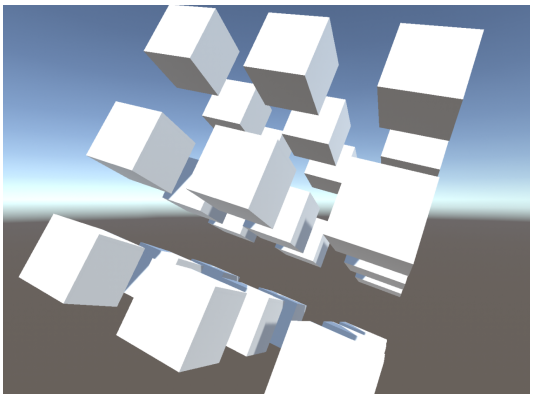

Fig. 10. Basic cube scene

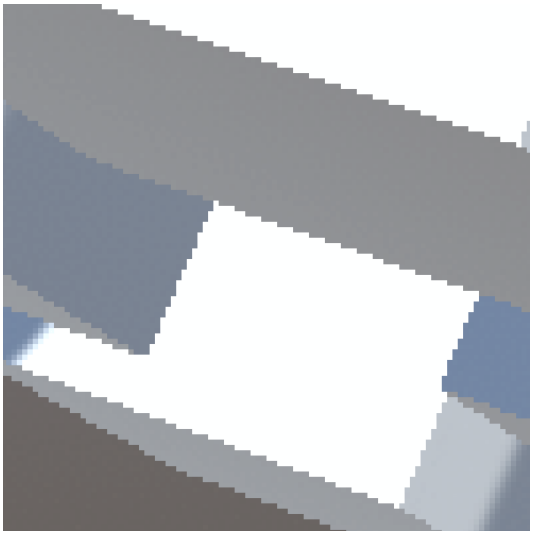

Fig. 11. 5x zoom on a portion of scene

Figure 10 shows the simple environment that was made, inspired by the testing environment created in the Playdead temporal solution[4]. A collection of cubes with some being occluded by others. Figure 11 shows a 5x zoom applied to the game view within Unity. By zooming in to the cubes it becomes much more apparent what the problem that is attempting to be solved with this implementation. Jagged edges are very apparent on lines of geometry that are not perfectly horizontal or vertical as the pixelated line at some point has to jump over to the next pixel in either direction. A stair-like jagged line is very apparent and especially during motion

of the camera can be very distracting and even sometimes results in a shimmering effect.

Next was to begin a solution to jitter the camera perspective and this was accomplished in a script attached to the camera object within the Unity project.

```
private float GetHaltonValue(int index, int base)
    {
        float fraction = 1.0f / (float)base;
        float result = 0.0f;

        while (index > 0)
        {
            result += (float)(index % base) * fraction;

            index /= base;
            fraction /= (float)base;
        }
        return result;
    }
private Vector2 GenerateRandomOffset()
    {
        Vector2 offset = new Vector2(
        GetHaltonValue(sampleIndex, 2),
        GetHaltonValue(sampleIndex, 3));

        if (++sampleIndex >= 16)
            sampleIndex = 0;

        return offset;
    }
```

The two above functions are required to calculate the offset that will be applied to the camera projection matrix. Sample index counts up to sixteen as the first sixteen samples of the Halton sequence (2,3) will be sufficient to create a distribution pattern that is suitable for this purpose. GenerateRandomOffset simply creates a vector that stores two values, the Halton value when the base is two and the value when the base is three and also increments the sample index up until sixteen where it will reset to zero. To calculate this value, we pass the current index along with the base values of two and three to the GetHaltonValue function. GetHaltonValue will return a Halton value for any base and the index of that sequence. A vector of two values both from the same index but with different base values will be returned when this code has completed which will act as the offset that we will use to shift the camera projection. This code was adapted from the pseudocode included in the implementation section of [6].

Next before the camera culls the scene, we will generate a new offset using

these functions. This offset is then used to create a new perspective projection matrix through a function adapted from Playdead's implementation [4] and this new perspective projection matrix becomes the current camera projection matrix and will effectively jitter the camera perspective as we have been aiming for. Each new frame it will shift to the next point in the sequence as the sample index is increased each time the offset is generated. Resulting from this code is that the image jitters in a seemingly random fashion as expected. It is important to save the offset in a global variable so that it can be accessed later as it will be required in the shader further down the line.

### *4.2 Initial resolve pass*

Now that the jittering is complete, the next step is to create a resolve pass that will effectively blend all of the previous frames into a history buffer. The current frame is blended with the history buffer and rendered to screen and then becomes the next history buffer.

An important feature of the Unity engine that is essential to this implementation are render textures, these are textures that can be rendered to and are typically used to make a camera render to a texture rather than directly to the screen [12]. We can use a render texture to be able to manipulate the output of the camera in order to achieve temporal anti-aliasing.

Render textures can be used in conjunction with the graphics function Blit, this copies a source render texture to a destination render texture with a material which often will contain a shader [13]. This allows the render texture to be used within a shader to be able to perform calculations on the GPU that determine the colour of the pixels that are rendered to the screen. The shader will perform the calculations necessary to implement the temporal anti-aliasing solution.

In the script attached to the camera, some objects need to be retrieved such as the camera itself and the shader that will be used. A material is created with this shader which can be used in Blit functions later. A render texture variable is declared which will be the history buffer that will be continuously updated on each frame.

An event function OnRenderImage [14] can be used to modify the camera's final image before it is displayed on screen. It is typically used for a full-screen post-processing effect and is where our blit functions will be executed. To begin, we check if a history buffer render texture has the same dimensions as the source camera render texture, if it doesn't, the history buffer texture is released. Then it will create a new blank render texture with the same camera render texture dimensions and copy it into the history buffer variable. A new blank render texture is also copied to the history buffer if the history buffer itself was empty.

Next is to pass two different variables through to the material so they can be used in the shader, the history buffer and the jitter offset. By passing through these variables into the material, it allows them to be accessed and used within the shader and can be used to achieve a resolve pass.

```
Graphics.Blit(source, temporary, material, 0);
Graphics.Blit(temporary, m_History);
Graphics.Blit(temporary, destination);
```

The above code are the main blits that are executed to achieve the desired result. Within the OnRenderImage function, which has two render texture parameters, source and destination. Source being the render texture from the camera before the final render and the destination being the render texture that is rendered to camera by the end of the function.

The first blit is the main function to operate the first resolve pass. The source render texture, from the camera, is copied into our previously created temporary render texture which will be blank. It is passed with the material which contains the shader. The resulting render texture that will be stored in the temporary render texture is the texture from the camera with the shader applied to it. This temporary shader is then copied into the history buffer which is a global variable so it can be reused when the function is called again on the next frame. It is also copied into the destination render texture which will be the final render texture which will be rendered to screen.

```
sampler2D _MainTex;
sampler2D _HistoryTex;
```

Now within the shader, we have our two main textures, main texture which is from the camera and the texture from the history buffer. These two textures are used within the fragment shader. This is a program that runs on every single pixel of the object or material on the screen which in this case is the camera render texture. This is where calculations are made to determine the colour of each pixel, and is executed for each of the millions of pixels on screen [15].

```
 float4 fragment(Varyings input) : SV_Target
{
    float2 uv = input.uv;
    float4 colour = tex2D(_MainTex, uv);
    float4 history = tex2D(_HistoryTex, uv);
    return lerp(colour, history, .9);
}
```

These are the most important steps within the shader and take place within the fragment shader. The first line retrieves the UV coordinate of the pixel that is currently being operated on within the texture. The next two lines are then sampling the current colour and the history colour from the main texture and the history texture respectively. In other words, we are sampling the colour of the pixel

in the same coordinate space in both the current frame for colour and the previous frame for history. Now that we have the colour of the pixel from both the current and previous frames, we use lerp which linearly interpolates between the two vectors [16]. The factor for now is 0.9 which means the colour that will be returned from the fragment shader is 0.9 through from colour to history so the history colour receives a larger weighted share of the colour that is returned. It is effectively blending a small amount of the current colour into the history buffer.

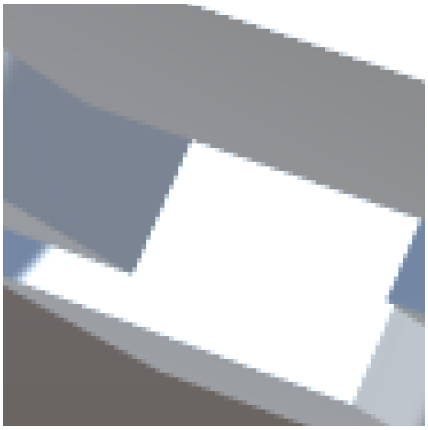

Fig. 12. 5x zoom with initial resolve pass

Figure 12 shows the result of the initial resolve. Each jittered frame is being blended together to create a smoother result with less jagged edges that looks noticeably better than the result from Figure 11. Although this result looks suitable stationary, significant problems arise when any type of motion is applied to the scene.

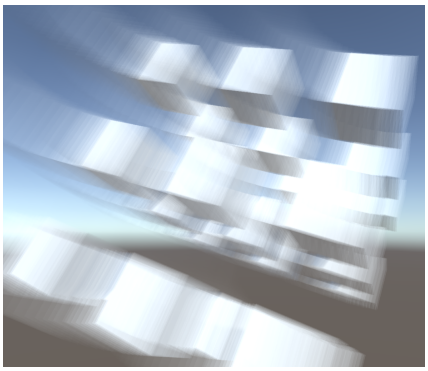

Fig. 13. Ghosting with camera motion

Figure 13 shows an example of the ghosting effect that is present in the solution currently which is similar to the one found in the earlier section in Figure 7. When motion is introduced, we are no longer sampling from the history texture in the correct location that corresponds to where the object is in UV space on the current frame. This can lead to sampling of a place with different lighting or even a different object completely. Reprojection needs to be implemented in order to correctly sample from the history texture in the place the pixel was located in the previous frame in comparision to the current frame.

### *4.3 Reprojection*

Reprojection can be achieved with the use of a motion vector texture in engines that support them. Unity supports the use of motion vectors which can be optionally enabled to render a motion vector texture if possible. This enables the rendering of another render pass from the camera after opaque render but before any image effects are applied. A full screen pass for camera motion is rendered first to calculate the screen space motion from the movement of the camera. Next another pass is rendered which effects moving objects to render object specific motion [17]. This texture which captures the screen space motion of objects per pixel from one frame to the next can be used for various image effects and most importantly can be used in temporal anti-aliasing solutions.

```
void OnEnable()
    {
        camera.depthTextureMode = DepthTextureMode.Depth
        | DepthTextureMode.MotionVectors;
    }
```

To begin implementing this motion vector into our solution, we first must enable the depth texture mode relating to motion vectors. This allows for the reading of a texture sampler which contains the generated motion vectors and can be used in image effects such as the shader that is currently being used.

```
sampler2D _CameraMotionVectorsTexture;
```

In the shader, the motion vector texture is accessed by this line which will return the motion vector, this can then be used in the fragment program within the shader.

```
 float4 fragment(Varyings input) : SV_Target
{
    float2 uv = input.uv;
    float2 motion = tex2D(
    _CameraMotionVectorsTexture, uv).xy;
    float4 colour = tex2D(_MainTex, uv);
    float4 history = tex2D(_HistoryTex, uv - motion);
    return lerp(colour, history, .9);
}
```

The following changes are made to the fragment shader program. A new vector called motion is created and returns the X and Y values returned when sampling the motion vector texture at the UV coordinates. This motion value represents the velocity of the pixel that is sampled at any given coordinate. It represents the amount the pixel has moved from the previous frame to the current one as it is the

offset of the previous and current positions of that pixel. Now that we know how much the pixel has moved between the current and previous frame this can then be used when sampling the history colour. By subtracting the motion from the UV coordinate, it is reconstructing the previous position of the pixel as it is reversing the motion that the pixel took between the two frames. Using this motion value, it can sample the pixel in the correct position in the history texture even where it may have moved.

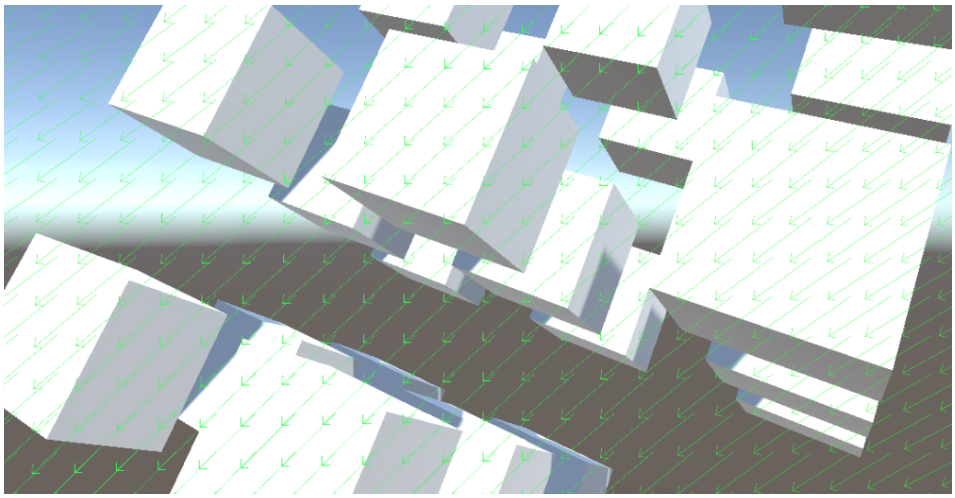

Fig. 14. Motion vector debug mode

To further illustrate how the motion vector texture works in Unity, a post processing debug mode can be enabled to demonstrate the effect fully. Figure 14 shows what the debug mode looks like when the camera is panning up and to the right. The green arrows represent the direction that each pixel has moved across in screen space between frames. Motion vectors are successfully tracking movement of pixels across screen space from one frame to the next.

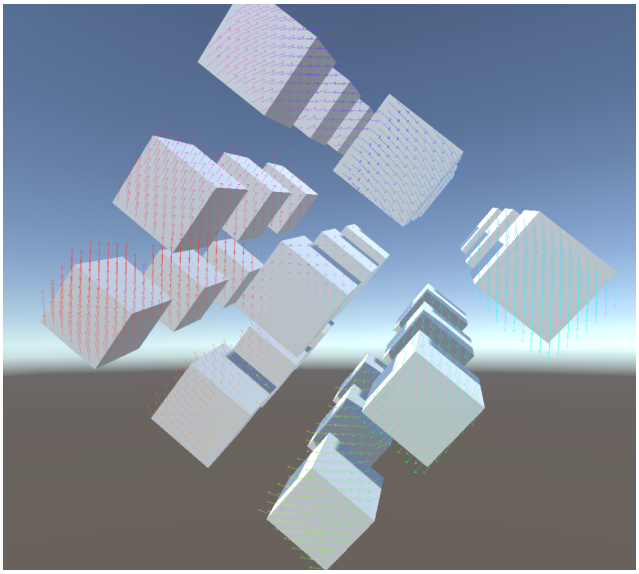

Fig. 15. Motion vector debug mode with rotation

Figure 15 shows an example of object motion being tracked rather than camera motion. In this image the camera is stationary however the cubes have a rotator script attached to them that rotates the object at a constant speed. The arrows represent the motion of each object with each motion being tracked independently. The outer cubes are further from the center and are travelling a larger distance on the screen space which is represented by the larger arrows compared to the arrows on the more inner cubes.

This solution has fixed the severe ghosting that is present when rotating the camera, however there are still issues with ghosting as result of disocclusion.

### *4.4 Neighbourhood clipping*

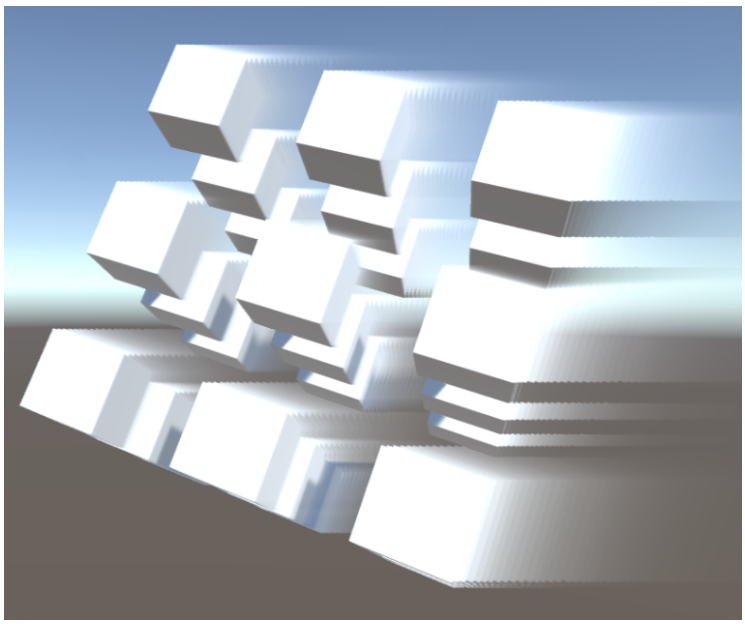

Fig. 16. Disocclusion causing ghosting

Figure 16 shows the effects that disocclusion has. By panning the camera side to side rather than simply rotating it, other portions of the scene become disoccluded and others are no longer occluded. The history sample cannot account for objects occluding each other between frames and therefore is invalid as it is sampling either an object that wasn't previously in front of another or another object is now in vision because the one in front has moved away and is no longer occluding it. Other factors may make the history sample invalid such as changes in lighting. This invalid sample is then blended into the history buffer and produces the ghosting artifact.

To remove the effect of ghosting, the invalid samples must be rectified so that if they diverge greatly they are adjusted and let through into the accumulation. To implement this, a 3x3 neighbourhood clipping solution is the best option as it is a pure colour space operation and can be applied just before the current and history colours are blended in the fragment shader.

```
float2 du = float2(_MainTex_TexelSize.x, 0.0);
float2 dv = float2(0.0, _MainTex_TexelSize.y);
```

The above code is added to the fragment shader program. To begin, the texel size of the render texture must be returned. The X and Y texel sizes equal one divided by the width and the height of the render texture respectively. The two floats DU and DV represent the size in UV coordinates required to move across to the next pixel in the X and Y directions. These values are required in order to sample colours in the 3x3 neighbourhood surrounding the pixel that is currently being operated on. By using these float values and modifying the UV coordinate of the sampled colour the pixels within the neighbourhood can be accessed and their colours sampled.

```
float4 topleft = sampleColour(_MainTex, uv - dv - du);
```

Above represents how the texel values are used in order to reach the surrounding colours in the 3x3 neighbourhood. The top left colour within the neighbourhood is found by subtracting from the UV coordinate in both directions in order to reach and sample in the correct space surrounding the current pixel. This operation is repeated for each space within the neighbourhood shown in Figure 8 on the right, other than the center space which is already defined as the UV coordinate of the current pixel.

```
float4 min = min(tl, min(tc, min(tr, min(ml, min(mc,
min(mr, min(bl, min(bc, br))))))));
float4 max = max(tl, max(tc, max(tr, max(ml, max(mc,
max(mr, max(bl, max(bc, br))))))));
float4 avg =
(tl + tc + tr + ml + mc + mr + bl + bc + br) / 9.0;

float4 min5 = min(tc, min(ml, min(mc, min(mr, bc))));
float4 max5 = max(tc, max(ml, max(mc, max(mr, bc))));
float4 avg5 = (tc + ml + mc + mr + bc) / 5.0;

min = 0.5 * (min + min5);
max = 0.5 * (max + max5);
avg = 0.5 * (avg + avg5);

history =
clip_aabb(min, max, clamp(avg, min, max), history);
```

Next is to find the minimum and maximum colour values found within this neighbourhood. By performing min and max operations on each value it will return the minimum or maximum colour value found within the entire neighbourhood. This minimum and maximum value can then be used later in order to define the bounds in which the sampled colour has to be within in order to be used in the accumulation stage. The average colour is then calculated by adding the different colours from each space in the neighbourhood together and dividing the result by the amount of spaces within the neighbourhood.

After this, as was demonstrated within the presentation by Karis [5] a plus shaped neighbourhood is also created in order to achieve a more rounded shape and a filtered appearance. The same steps as the first three lines are repeated however this time the corner spaces are removed. The following three lines then simply add the minimum, maximum and average colours together for both the 3x3 neighbourhood and the plus shaped neighbourhood together and then divide them in half in order to reach the halfway point between the two seperate values.

History colour is then updated with a call to the clipping function which has parameters for the minimum value, the maximum value and a clamped colour value from the neighbourhood and the already sampled history colour.

```
float4 clip_aabb(float3 aabb_min, float3 aabb_max,
float4 p, float4 colour)
    {
        float3 center = 0.5 * (aabb_max + aabb_min);
        float3 extents = 0.5 * (aabb_max - aabb_min);

        float4 distance = colour - float4(center, p.w);
        float3 repeat = abs(distance.xyz / extents);

        float repeatmax  = max(repeat.x, max(repeat.y,
        repeat.z));

        if (repeatmax > 1.0)
            return float4(center, p.w)
            + distance / repeatmax;
        else
            return colour;
    }
```

The clip function is an adaptation of the function as explained in the Playdead presentation [4]. Clipping is performed only towards the center of the AABB bounding box as the operation is faster than clipping towards the edge of the box. The distance is calculated as the difference between the sample colour and the colour at the center of the bounding box. The value repeat refers to the distance the sample colour is from the center of the box. A repeat value of one implies that the colour sits on the edge of the box, less than one would mean the colour falls within the box

and greater than one would mean it is beyond the extents of the box. If the colour sits outside of the bounding box it is clipped so that the colour is not greater than the extents of the box. If the colour falls within the box, the colour is returned as it fits within the extents of the neighbourhood colours.

The result of this operation is that colours that exceed the range of colours found within the surrounding neighbourhood are not discarded but are instead adjusted by clipping towards a box that represents the range of valid colours. This means that colours sampled from the history texture cannot deviate far from the current neighbourhood of colours and are instead adjusted to fit within the extents of the colours found within the neighbourhood surrounding the current pixel.

# 5 Evaluation

The evaluation of the implementation will consist of two main parts. The first will be analysing the visual impact of the temporal anti-aliasing solution with comparative images to the native output to see the benefits and some drawbacks to the solution. These comparisions will compare the visual difference between the native and temporal anti-aliased image in particularly challenging scenarios in which issues can often arise with temporal solutions.

The other section will discuss the performance of the solution to see if the temporal solution achieves the goal of creating a lightweight anti-aliasing solution as the accumulation buffer allows for samples to be gathered over multiple frames rather than just in one.

Images captured unless specified otherwise are captured at a game resolution of 1920x1080. Testing was completed on a personal computer with the following specifications:

- CPU: AMD Ryzen 7 3700X 8 Cores 16 Threads 3.60GHz Clock 4.4GHz Max Boost
- GPU: NVIDIA GeForce RTX 2070 Super with 8GB of VRAM
- RAM: 16GB DDR4 3200Mhz

All post processing effects are disabled other than the temporal related script and shaders that are being tested. Native refers to the output from the game view with no anti-aliasing applied at all and TAA refers to the temporal anti-aliasing solution being active. Images are taken within the 3D environment found from the Unity Asset Store named Flooded Grounds [18]. The script and shaders developed for this solution were simply attached to the camera object within this scene. This provided an environment to test the solution in a scene that resembles one that would be found within a commercial video game release to assess the success of the implementation in the scenario in which it would be used.

Fig. 17. Native vs TAA; 5x zoom; General test.

Figure 17 depicts a scene from the game environment used to analyse the effectiveness of the temporal solution. This particular example is a basic example of the solution at work and is static. It can be seen in this image that the temporal

anti-aliasing solution is very effective at removing jagged edges from the diagonal edges on the house. The roof in particular as the background behind it is a vastly different colour so it stands out as having visible jagged edges in the native image but the TAA cleans up the image significantly. The wooden beam going across the house at the bottom of the image also looks noticeably cleaner. Since TAA is a screen space operation and works on the whole image rather than just the vertices of geometry, even jagged edges within the texture of the wooden planks on the house are smoothed out.

Fig. 18. Native vs TAA; 3x zoom; Tree foliage test.

An important area to analyse for anti-aliasing solutions are foliage such as grass and trees. They tend to be comprised of small objects on the screen such as strands of grass or leaves on trees and when rendered natively can appear heavily jagged and can shimmer in motion heavily. Figure 18 shows an example of a tree within the environment with the solution on and off. It does a good job smoothing out the jagged edges on the outermost leaves. Apparent in the native image are small blue pixels in the center of the tree where the leaves don't cover a very small portion of the sky and result in flickering blue dots within the tree as the leaves sway slightly or the camera moves position and is distracting. TAA smooths out these blue pixels as the jittering gains more information about the sub pixel detail and more of the green colour within the leaves is accumulated into the history buffer and removes the small blue pixels.

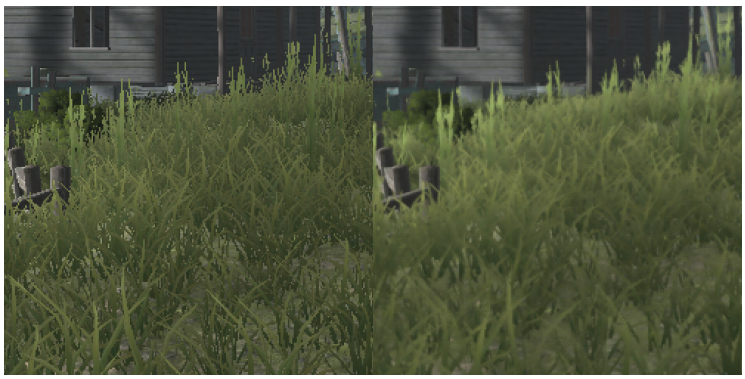

Fig. 19. Native vs TAA; 3x zoom; Grass foliage test.

Figure 19 shows a patch of grass outside of the house. There is a stark difference between both images, the native image looks sharper with each strand of grass

standing out more. The temporal image loses the sharpness and is blurrier as a result but the jagged edges are reduced severely. The largest difference being the taller strands of grass towards the back that stand taller than the rest. In the native image, this grass is not being rendered in full due to the resolution being too low to be able resolve these pixels properly. The strands of grass are so small in the image space that they sometimes do not cross the center point of the pixel and thus do not get rendered. This results in floating green pixels where the grass is thinner leaving gaps in the strands. The TAA version however does a better job at resolving these fine lines of grass with the accumulation buffer gathering higher amounts of sub pixel details across multiple frames with jittering aiding the accumulation.

Fig. 20. Native vs TAA; 2x zoom; Fast motion test.

Figure 20 shows a comparision of two images taken with a sphere object moving at a very fast speed from the left side of the screen towards the right with a screenshot taken when the sphere is in the center of the camera view. By comparing the two images there is a visible banding artifact on the top left side of the sphere in the TAA version however this is a very minor artifact that is hardly visible in motion. The boat is slightly blurry due to the need for data to re-accumulate in the history buffer as the sphere moves so fast and occludes the boat temporarily. This example particularly shows the importance of history rectification in order to reduce ghosting trails behind objects in motion.

Fig. 21. TAA; 2x zoom; Fast motion test; Neighbourhood clipping disabled.

Figure 21 is the same scenario found in Figure 20 however this time the sampled history colour is no longer being clipped to the neighbourhood. The result of this is a very large amount of ghosting, showing how essential the history rectification is to the temporal solution as ghosting this severe would make the solution unusable. The

area with the boat becomes disoccluded when the sphere moves beyond it, however when the pixels in that area sample from the history texture they encounter invalid information from when the sphere was previously occluding it. By clipping the history colours to the current pixel neighbourhood, it adjusts the history colour to more resemble the colours found within the current neighbourhood and not the colours of the sphere that has already moved past it.

### *5.1 Performance*

Performance data was collected through Unity using the included profiler. This contains information such as the CPU and GPU usage and time and allows the user to see statistics for every single frame.

| Anti-aliasing | Render Resolution | | |
|---|---|---|---|
| | 1920x1080 | 2560x1440 | 3840x2160 |
| No AA | 0.40ms | 0.52ms | 0.85ms |
| TAA | 0.57ms | 0.79ms | 1.56ms |
| Cost | 0.17ms | 0.27ms | 0.72ms |

Table 1
Average GPU time for native and TAA at different resolutions in cube scene.

Table 1 contains data relating to the performance on the GPU that the temporal anti-aliasing solution costs. This data was captured from the basic cube scene in order to minimise external influences affecting the GPU performance in a larger complex scene. At 1920x1080 resolution, the temporal implementation costs 0.17ms of GPU time, in other words, each frame takes 0.17ms longer to render on the GPU with the temporal anti-aliasing enabled compared to it disabled. The cost of the temporal anti-aliasing seems to have correlation with the amount of pixels rendered on screen. A resolution of 1920x1080 renders around 2.07 million pixels whereas 3840x2160 (4K) is 8.29 million pixels, four times the amount. A cost of 0.17ms multiplied by four is 0.68ms which is a similar cost (0.72ms) when temporal anti-aliasing is operated on a 4K resolution. It makes sense for this to be the case as the shader used in the solution operates on each pixel individually in the fragment shader program and the operation on each pixel should take the same amount of time.

With modern games adapting higher resolutions like 4K, 0.72ms of extra GPU time when using the temporal anti-aliasing solution is not highly expensive. For a game to run at 60FPS (frames per second) it must render a new frame every 16.67ms. If this solution was implemented in a game targeting 60FPS at 4K, 15.95ms would be the maximum amount of time that the GPU can perform all the operations each frame other than the temporal pass. Performance is reduced by 4.3% when using this temporal anti-aliasing solution which is not very expensive.

It must be added that CPU time was also attempted to be recorded as the solution as a whole is not executed solely on the GPU. The script itself is operated by the CPU and will have a performance impact on the time the CPU takes to render

each frame however in the cube scene the CPU time fluctuated greatly making it hard to take an average and the average that was recorded was not consistent across tests whereas the GPU time was consistent.

The cube scene is a very low complexity scene which is not indicative of typical real-time performance as framerates were sometimes over 600, however it was the best way to measure the performance of the GPU completely isolated from other factors. The Flooded Grounds [18] environment was used again which is more representative of real-time rendering loads in order to measure the performance impact in a complex environment. Measuring performance in this environment was achieved through the program MSI Afterburner and recording a benchmark of performance across a thirty second time span as the player walks down a path from one end to the other. This was done with temporal anti-aliasing enabled and disabled and the average framerate from the benchmark run was returned.

| Anti-aliasing | Render Resolution | | | |
|---|---|---|---|---|
| | 1920x1080 | 2560x1440 | 3840x2160 | 7680x4320 |
| No AA | 89.8 | 89.4 | 84.6 | 54.0 |
| TAA | 89.1 | 87.5 | 80.6 | 46.0 |

Table 2
Average framerate across benchmark runs in flooded grounds.

Table 2 shows the results of the benchmark runs across the more complex environment. For a 1920x1080 resolution, the average framerate decrease from using TAA is negligible with only 0.7 less, 2560x1440 is still small but the difference grows to 1.9FPS less and for 3840x2160 it is 4FPS lower. This shows that the temporal solution has a small impact on performance but is cheap so framerates will not suffer in a more game-like scenario. However, it is clear that the lower resolutions being very similar without TAA enabled shows that the framerate in those cases were limited by the CPU. The CPU usage in the environment was quite high and could affect results however it is still clear that temporal anti-aliasing was still affecting performance even at lower resolutions where the CPU was the limiting factor as the performance always reduced, however small an amount. These results further conclude that temporal anti-aliasing is more costly as the resolution increases as the higher resolutions result in a greater reduction in framerate.

For this scenario an 8K resolution (7680x4320) was even tested even though it is currently an unrealistic resolution target in most games. This was because with a framerate of 84.6 at 4K without TAA, it was similar to the previous two lower resolutions meaning that the CPU could still be a limiting factor. The much lower framerate of 54.0 without TAA at 8K showed the scenario in an environment where the GPU was the limiting factor and leads to an average framerate loss of 8.0 when enabling TAA which is far more significant than the other resolutions, however, due to the unrealistic amount of pixels being operated on, it isn't exactly indicative of typical performance loss.

# 6 Conclusion

By the end of the project a temporal anti-aliasing solution was created which contained techniques including projection matrix jittering, reprojection and history rectification through neighbourhood clipping. The aim of the project was met as the implementation of these techniques was successful with all of the objectives being met. The resulting output from testing provided an anti-aliased image with jagged edges removed and minimal artifacting as a result of motion. Jittering accumulated more sub-pixel details from static images into the history buffer and blended with the current frame to create an anti-aliased image. Reprojection successfully eliminated ghosting artifacts especially those caused by camera rotation and movement. Neighbourhood clipping was successful in preventing further ghosting trails caused by occlusion and disocclusion.

Testing showed that performance impacts from the solution were mostly minimal. At lower but still common resolutions such as 1080p and 1440p the difference in performance was very small and GPU times were only increased by 0.17ms and 0.27ms respectively. At 4K resolution which is becoming more common the performance impact was greater at 0.72ms but still not expensive. In a real-time typical game environment scenario this translated to at most 4 frames per second reduction at 4K which is more than acceptable. Lower resolutions had negligible framerate reductions. The goal to create a solution that is light on performance was sufficiently met.

## *6.1 Improvements*

Although the project itself was a success, there are many ways in which it could be improved if attempted again or given more time. First of which is the sharpness of the image. Temporal anti-aliasing is known for a softer image than native rendering however sharpening techniques involving bicubic filtering can be applied to the final image in order to reduce the blurriness, this could have been applied to this implementation as the output is perhaps too soft.

Although the generation of a motion vector texture is present in Unity, it is not applicable to every game engine or API and therefore makes the solution presented limited in terms of adaptability across to other engines. A method of reprojection such as using depth buffers and projection matrices to reconstruct world positions in order to sample from the history texture in the correct location could have been implemented in order to create a more general solution.

Another thing that could have been explored in this project is a motion blur fallback system like the one mentioned in the Playdead presentation [4]. At low resolutions and framerates, history fragments can still remain if they are left behind by neighbourhood clipping. Motion blur can be applied to the fast moving fragments on output only in order to conceal the artifacts.

Finally, the testing of the implementation could be improved. Temporal anti-aliasing could be compared with other anti-aliasing techniques further but only one other is built into Unity itself and time was not available to recreate testing again. The testing environment could also be changed. The environment had an unexpected CPU usage that could have influenced the results.

# References


[1] Lei Yang, Shiqiu Liu, and Marco Salvi. A Survey of Temporal Antialiasing Techniques. *Computer Graphics Forum*, 2020.

[2] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Tiago Sousa. Anti-aliasing methods in cryengine 3. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Lasse Jon Fuglsang Pederson. Temporal reprojection anti-aliasing in inside. Game Developers Conference, 2016.

[5] Brian Karis. High quality temporal supersampling. SIGGRAPH, 2014.

[6] James Heald. Halton sequence - wikipedia. https://en.wikipedia.org/wiki/Halton_sequence, 2022. Accessed: 14/05/22.

[7] The Code Corsair. Temporal aa and the quest for the holy trail. https://www.elopezr.com/temporal-aa-and-the-quest-for-the-holy-trail/, 2022. Accessed: 14/05/22.

[8] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, page 25–35, Goslar, DEU, 2007. Eurographics Association.

[9] LearnOpenGL. Learnopengl - coordinate systems. https://learnopengl.com/Getting-started/Coordinate-Systems, 2022. Accessed: 14/05/22.

[10] Malan, Hugh. Realtime global illumination and reflections in dust 514. https://advances.realtimerendering.com/s2012/CCP/Malan-Dust_514_GI_reflections(Siggraph2012).pptx, 2012. Accessed: 23/05/22.

[11] Sousa, Tiago. Graphics gems from cryengine 3. https://www.slideshare.net/TiagoAlexSousa/graphics-gems-from-cryengine-3-siggraph-2013, 2013. Accessed: 23/05/22.

[12] Unity Documentation. Unity - scripting api: Rendertexture. https://docs.unity3d.com/ScriptReference/RenderTexture.html, 2021. Accessed: 26/05/22.

[13] Unity Documentation. Unity - scripting api: Graphics.blit. https://docs.unity3d.com/ScriptReference/Graphics.Blit.html, 2021. Accessed: 26/05/22.

[14] Unity Documentation. Unity - scripting api: Monobehaviour.onrenderimage(rendertexture,rendertexture). https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnRenderImage.html, 2021. Accessed: 26/05/22.

[15] Unity Documentation. Unity - manual: Custom shader fundamentals). https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html, 2021. Accessed: 29/05/22.

[16] Unity Documentation. Unity - scripting api: Vector4.lerp). https://docs.unity3d.com/ScriptReference/Vector4.Lerp.html, 2021. Accessed: 29/05/22.

[17] Unity Documentation. Unity - scripting api: Depthtexturemode.motionvectors). https://docs.unity3d.com/ScriptReference/DepthTextureMode.MotionVectors.html, 2021. Accessed: 31/05/22.

[18] Sandro T. Flooded grounds — 3d environments — unity asset store). https://assetstore.unity.com/packages/3d/environments/flooded-grounds-48529#publisher, 2019. Accessed: 02/06/22.