Screen Space Ambient Occlusion

by Andrew Hamilton (Orange451) and Michael Parkin-White (MishMash95)

Contents:

1) Introduction:

What is screen space ambient occlusion?

Screen space ambient occlusion is a post-process effect which attempts to approximate shadows produced by nearby objects in a scene. A simple example of this would be darkening of crevasses in objects, as well as the illusion of absence of light found between two objects near each other.

In games, this effect aims to improve the visual realism of scenes by increasing visual depth of objects. The shader works in screen-space meaning it is a post process effect which makes use of (hopefully) already existing per-fragment depth and normal data.

Screen space ambient occlusion is still quite a high-end effect, and will often only run be feasible on high-end machines if you plan on maintaining 60 fps.

Using SSAO in Gamemaker:Studio :

It is worth noting that SSAO is not an immediately simple effect to integrate into your project. It will require that certain things are setup in a certain way. This is simply to make the effect as efficient as possible. There are a few major steps in the process of creating the effect that you need to consider, and ideally understand in order to implement the effect successfully:

- (A) Rendering your scene geometry to diffuse, depth and normal buffers.
- (B) Generating the ssao buffer using the previously generated depth and normal buffers.
- (C) Blurring the ssao buffer to remove unwanted noise.
- (D) Combining the ssao with the diffuse and rendering it to the scene.

All of these will be covered in detail in the implementation guide. I will write a guide for both basic and advanced use, here is a simple criteria you can use to decide which one you need:

- **Basic use:** If you are only looking to add ssao to a simple 3D project which does not implement other shaders.
- **Advanced use:** If you are looking to combine ssao into a project which already features other effects such as lighting, shadows, normal mapping etc;

As always, if you get stuck implementing the ssao, we can help guide you on your way. Please just send one of us a message, or leave a reply on the Marketplace topic on the GMC.

2) Understanding: Screen space ambient occlusion:

How does SSAO work?

SSAO works by running a small program called a fragment shader for every pixel on the screen. It is a post-process effect meaning it is performed after all other core rendering has been done. For each pixel, an occlusion value is determined by performing a number of samples of neighbouring pixels. This allows an occlusion value to be constructed for that pixel to determine how much that pixel is occluded by other pixels.

This relies on both normal (i.e the way a mesh's triangle is facing) data and depth data (How far away the point is from the camera) for each pixel.

3) Adding SSAO to your project

Simple guide:

In order to add SSAO to your project, you will need to perform a few steps:

1) Import the SSAO shader asset from the marketplace into your project.
2) You will need to move all of your object's 3D rendering to the **world_draw()** script.
   The easiest way to do this is to put all of your current objects draw code into a user_defined event and have code which looks like the following:

   ```
   /// world_draw
   with( objPlayer) { event_user( 0 ); }
   with( objEnemy) { event_user( 0 ); }
   d3d_model_draw( global.LevelModel, 0, 0, 0, background_get_texture( levelTexture ));
   ```

   The one condition here however is that you CANNOT use any shader-based rendering here. If you do need to implement ssao into a project which already uses shaders, please see the **advanced guide**.

3) Setting up the camera: By default, the SSAO shader comes with a basic 3D camera, this can be easily modified. The easiest way to do this is to either just use our camera, or to directly change the **camera_set_projection()** script to make use of your own camera.
   Please note, certain other places also make use of the camera's variables. Namely:

   obj_camera.cameraFar and **obj_camera.cameraNear** are both referenced and will need to be replaced if you are using your own camera outright.

   You can go to the scripts -> Search in scripts in the game maker menu bar to find all occurrences of **obj_camera**, and replace them with your own.

   The ssao scripts will call **camera_set_projection**() at any time where a 3D projection needs to be rendered, so your camera projection MUST be defined in here, rather than where you might have been doing it before.

4) **obj_ssao** is the only important object that you need to keep, **obj_world** only performs loading in the demo, so you can delete that if you want. (So long as you're loading is happening somewhere else.)

5) The SSAO_RENDER script performs the majority of the tasks required. The required buffers are rendered, the ssao calculated and then the result blurred. This process is mostly locked down and you will not need to change this code.

6) The final step is to actually display the result on the screen. In SSAO_DRAW, the SSAO is combined with the diffuse (plain 3D render) buffer in a shader then drawn to the screen.

7) Finally, you can strip out any of the debug code you do not need. All debug code is labelled debug, so that code is all safe to remove.

Advanced guide:

If you already have a number of complex shaders embedded in your project, then it is potentially going to be quite difficult to implement SSAO into your project. This guide is going to cover a number of practises you can try to follow to make this process as easy as possible.

So, the SSAO shader is going to need access to the following things:

- A **Depth buffer**. (Pre-configured depth buffer is a 24-bit depth buffer.) (This will take the form of a surface.)
- A **Normal buffer.** (Pre-configured normal buffer is a view-space normal buffer.)(As a surface.)

This is all you actually need to generate the SSAO, if your game does not have these already, you can just use the ones provided. The ssao buffer is then dealt with independently when blurred.

The other major thing you will need to change is the diffuse buffer setup. Currently, there are multiple paths the shader can take when it comes to rendering the buffers. The most optimal case is using **render_gbuffer_ddn()**, which will render all 3 buffers in a single pass using **HLSL9's Multiple Render Targets** (MRTs).

If you are already rendering your scene to a diffuse buffer (The diffuse buffer is just a surface which stores the plain-rendered scene.), then you can instead use the **render_gbuffer_dn()** which will just render the depth and normal buffers. (**GLSL ES** equivalents (render_diffuse_buffer, render_depth_buffer, render_normal_buffer) also exist if you are not using a windows platform.)

IT IS IMPORTANT THAT YOUR CURRENT 3D SCENE IS RENDERED TO A DIFFUSE BUFFER RATHER THAN STRAIGHT TO THE SCENE, AS IT IS NEEDED IN THIS FORMAT IN ORDER TO EASILY COMBINE WITH THE SSAO LATER ON.

Ideally, you want to set up your game in such a way where you are not rendering directly to GM's application surface during your 3D pipeline process. This is because you want to be able to re-use what you already have wherever possible, and not commit anything to the final surface until everything has been done.

If you have all this, it should now be relatively simple to get the SSAO into your game. At this point, at the end of your draw process, you should have access to:

- A diffuse buffer containing everything that you previously wanted to render, I.E lighting effects, shadows etc.
- The SSAO buffer

These two can now be combined with a simple multiply shader and rendered to the screen directly. In our demo, we do this in **SSAO_DRAW** using **sh_post_combine** which takes the two buffers, combines them and renders them to the screen. This is done in the draw_gui event, as both of this are flat surface textures.

In conclusion:
- The easiest way to achieve the best implementation is to just render what you already have to a surface, then generate the ssao buffer, and combine the two at the end and render both to the screen.

Notes:
- We disable the default application surface rendering in this, as its not needed. It does not give us the control needed as everything can then just be rendered to the screen in the draw gui event.

4) Customisable settings and optimisation

The SSAO shader has a number of customisable settings which can be used to change the visual appearance the quality of the effect. Whilst settings can be increased to improve the overall visual fidelity of the effect, it is worth noting that some of these settings may have more of an impact on performance than others.

Equally, the quality can be decreased in order for a performance gain. SSAO is still a high end effect so the shader has been optimised for the average use case at a good looking setting. The quality drop-off therefore may be quite quick when some of the settings start getting lowered.

Here is a list of all the settings you can modify, and recommended values:

SSAO SAMPLE COUNT:

Increased sample count increases accuracy of approximation at the cost of having to perform more iterations per-fragment.

Variable: global.SSAO_SAMPLES
Default: 16 (Recommended)
Alternative settings:

- 8 - Best optimal setting for increased performance
- 16 - Good average setting
- 32 - High-end setting to increase accuracy of approximation
- < Any higher value – diminishing returns >

Note: When modifying, this needs to be changed in SSAO_INIT, and in the fragment shader in sh_post_combine.

SSAO RADIUS:

The radius affects the maximum distance at which nearby objects are sampled. A higher radius will improve the result of occlusion on larger objects, however the accuracy will then decrease for finer object detail. This is mostly a preference. A very large radius will have an impact on performance however due to localised texture caching in GPU shader unit memory.

Variable: global.SSAO_RADIUS
Default: 5.0
Alternative Settings:

- 1.5 - A very fine radius which can highlight the detail of small objects.
- 3.5 - A mid setting to get the best of large and smaller detail
- 5.0 - An overall, good looking setting for powerful AO.

Note: The SSAO Radius may need to be changed if the scale of your scene differs from the scale in our demo. The scale used in our demo is approximately 16 units = 1.7m (Average height)

SSAO STRENGHT:

Strength darkens the overall produced AO by multiplying each occlusion sample. This can be used to control the base darkness of the AO. Generally high strengths can look messy, so this value is best kept low and subtle to make slight visual modifications to your desired result.

Variable: global.SSAO_STRENGTH
Default: 0.05
Alternative settings:

- 0.05 to 0.20 - Lower is generally better
- 0.40 to 0.50 - This is quite a harsh setting and does not look that good, but you may want it.

SSAO POWER:

Power provides a different way of adjusting the darkness of the AO. Power works by contrasting the final AO result. This allows the smooth gradient to be maintained and therefore power generally provides a much more pleasing look, even at higher settings.

Variable: global.SSAO_POWER
Default: 16.0
Alternative settings:

- 3.0 - For a nicer, subtle AO. This looks good with lower resolution ssao buffers.
- 16.0 - A middle setting
- 32.0 - A more intense setting for increased AO influence on scene lighting.

SSAO RESOLUTION

The SSAO Resolution controls the overall accuracy of the AO. Higher resolutions look better, however the performance cost at a higher resolution increases much faster due to the vast increase in number of pixels processed. The value of this acts as a multiple to the screen resolution.

Variable: global.SSAO_RESOLUTION
Default: 1.0
Alternative settings:

- 0.5 - Half resolution ssao_buffer, lower quality, but quite a bit faster
- 1.0 - Default, looks good and performs well.
- 2.0 - Increased visual accuracy at the cost of increased processing.

SSAO BLUR PASSES

The ssao result is blurred after it has been first processed to remove any noise left over from the ssao generation. By default, this is an advanced bilateral blur and so increasing the samples can cause a decrease in performance.

Variable: global.SSAO_BLUR_PASSES
Default: 2
Alternative Settings:

- 0 to 1 - No/little blur, you will still see noise at these settings
- 2 - Ideal blur, will never really need more than this.
- 4 + - May be needed with high radius + strength combo’s to smooth result.