Dynamic Radial Masks

Shader extension for Unity

Volume #1

Copyright © 2020 Amazing Assets

amazingassets.world

Contents

1 Tutorial

This chapter describes what is **Dynamic Radial Masks** using simple step-by-step guide. All steps here are described as easy as possible to give overall overview of this asset and how it works. Each part of this chapter is described in more details later in this file.

For creating **Dynamic Radial Mask** (DRM) effect it is required to follow three simple steps:

1. Create DRM class file - ***cginc*** file.
2. Call DRM execution method inside a shader.
3. Update DRM visual properties from script.

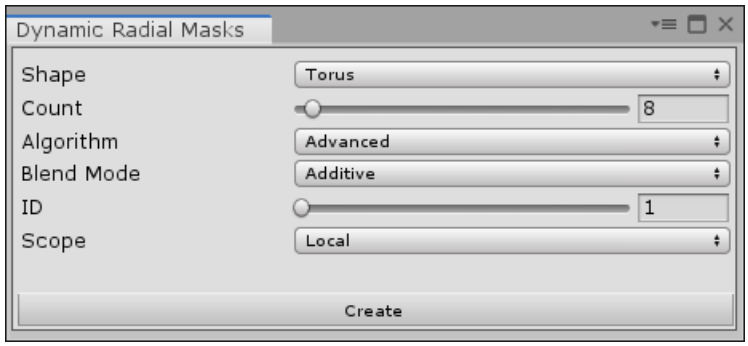
Open **Tutorial** scene (**Amazing Assets\Dynamic Radial Masks\Example scenes\Tutorial** folder). It contains simple Plane object with an unlit shader and some helper objects explained later in this tutorial. We will added DRM effect into this unlit shader.

Note, this tutorial explains shader building process for hand-written shader (Step 2a), for Unity Shader Graph (Step 2b) and Amplify Shader Editor (Step 2c). Before continue reading choose desired shader building tool and assign appropriate shader to the Plane game objects material. Shaders are in the same folder as **Tutorial** scene.

Step 1 – Creating DRM class file - *cginc* file

Before using DRM inside a shader it is required to generate ***cginc*** file. This file contains description of the DRM class with all variables and methods needed for calculation and visualization of a mask.

- Open DRM editor window from: Menu -> Window-> Amazing Assets -> Dynamic Radial Masks.



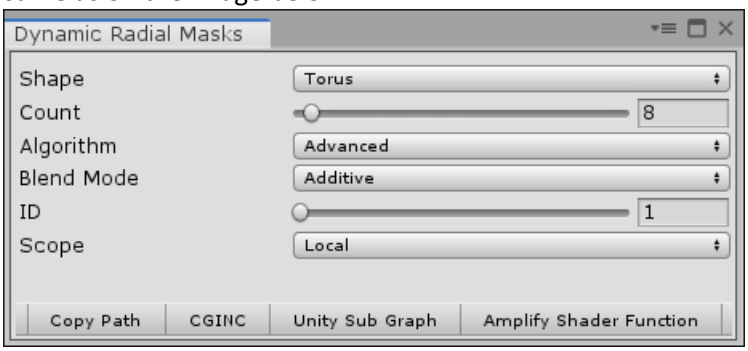
- Set properties as on the image above and press on **Create** button.
  All required files now are generated and are ready to be used in shaders. Do not close window, it will help us later.

Note, editor window is explained in details in chapter: Editor Window

Step 2a - Implementing DRM into hand-written shader

Basic knowledge of shader programming is required here.

- Make sure Plane's game object material is using **Tutorial (Hand Written)** shader.
- Open shader file in any text editor.
  This is very simple shader, calculating vertex world position and returning 0 value from a fragment shader. We will add mask calculation here.
- To use DRM inside a shader, first we have to include path to the ***cginc*** file where it is described. DRM editor window helps to navigate to the required file. Open it and make sure settings are the same as on the image below.



- Press on **Copy Path** button.
  Now path to the required ***cginc*** file is copied into the keyboard memory.
- Inside our unlit shader, below `#include "UnityCG.cginc"` (approximately line: 19) add new `#include` and paste text from the keyboard.

```
#include "UnityCG.cginc"
#include "Assets/Amazing Assets/Shader Extensions/Dynamic Radial Masks/Shaders/CGINC/Torus/8/Advanced/DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local.cginc"
```

- In pixel shader call **DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local** method and assign its value to the mask variable.
  Arguments for the DRM method is vertex world position (Absolute) and noise size (0 in our case).

```
float mask = DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local(i.worldPos, 0);
```

> Note, DRM method name is exactly the same as ***cginc*** file name where it is described. In our case it is **DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local.**
> More details about DRM class names is described in Dynamic Radial Mask chapter.

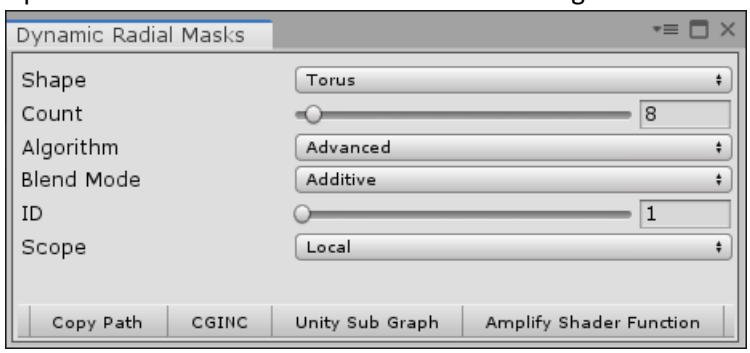
- Save shader file. It is fully functional now.

Step 2b – Implementing DRM into Unity Shader Graph

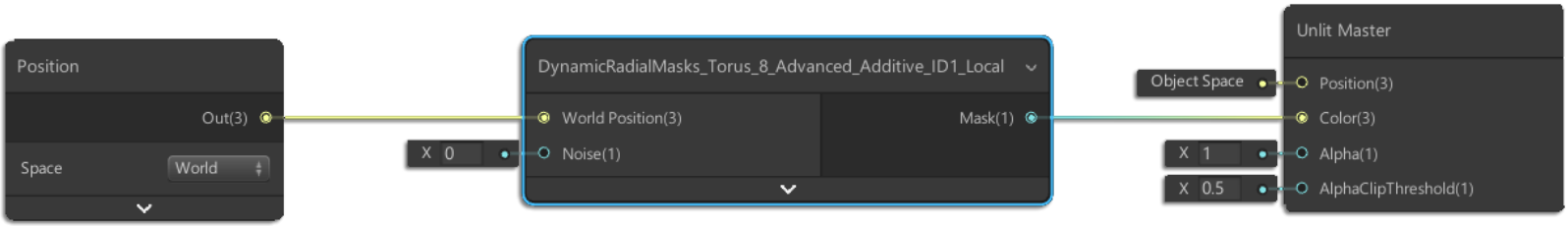
To use custom nodes inside Unity Shader Graph, make sure Unity editor version is 2019.2 or newer and project uses scriptable render pipeline.

- Make sure Plane's game object material is using **Tutorial (Unity Shader Graph)** shader.
  This is very simple unlit shader without any nodes. We will add mask calculation here.
- When we have created ***cginc*** file using DRM editor window, custom node for Unity Shader Graph also was generated. All we have to do is just drag and drop that node inside our shader.

- Open DRM editor window and make sure settings are the same as on the image below.

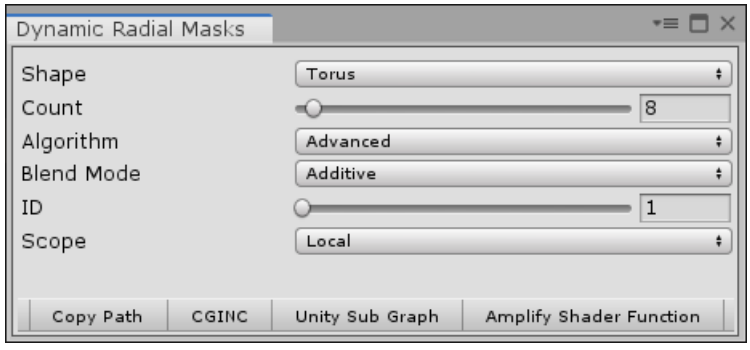


- Press on **Unity Sub Graph** button.
  Unity will navigate Project window to the folder where node file is and highlights it.
- Drag and drop that file inside shader editor.
- Connect **Mask** output value from this node to the **Color** property of the Unlit Master node and provide vertex's **absolute world position** data to the DRM node. Noise property is optional here and in this case leave it to 0.
- Save shader file. It is fully functional now.



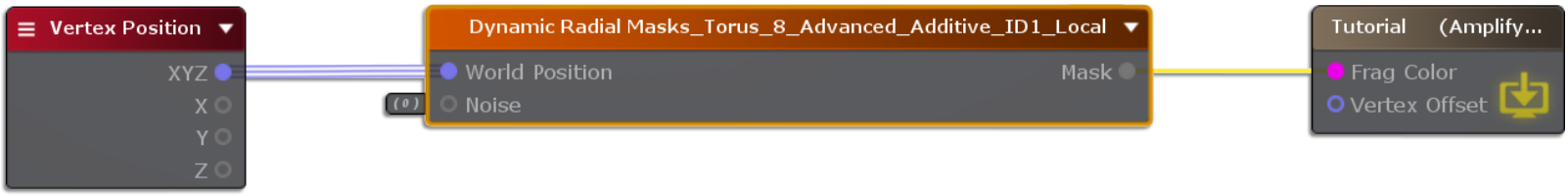
Step 2c – Implementing DRM into Amplify Shader Editor

- Make sure Plane's game object material is using **Tutorial (Amplify Shader Editor)** shader.
  This is very simple unlit shader without any nodes. We will add mask calculation here.
- When we have created ***cginc*** file using DRM editor window, custom node for Amplify Shader Editor also was generated. All we have to do is just drag and drop that node inside our shader.
- Open DRM editor window and make sure settings are the same as on the image below.



- Press on **Amplify Shader Function** button.
  Unity will navigate Project window to the folder where node file is and highlights it.

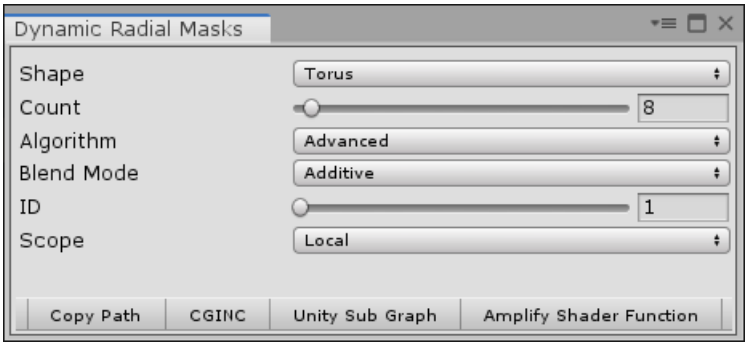
- Drag and drop that file inside shader editor.
- Connect **Mask** output value from this node to the **Frag Color** property of the Master node and provide vertex **world position** data to the DRM node. Noise property is optional here and in this case leave it to 0.
- Save shader file. It is fully functional now.



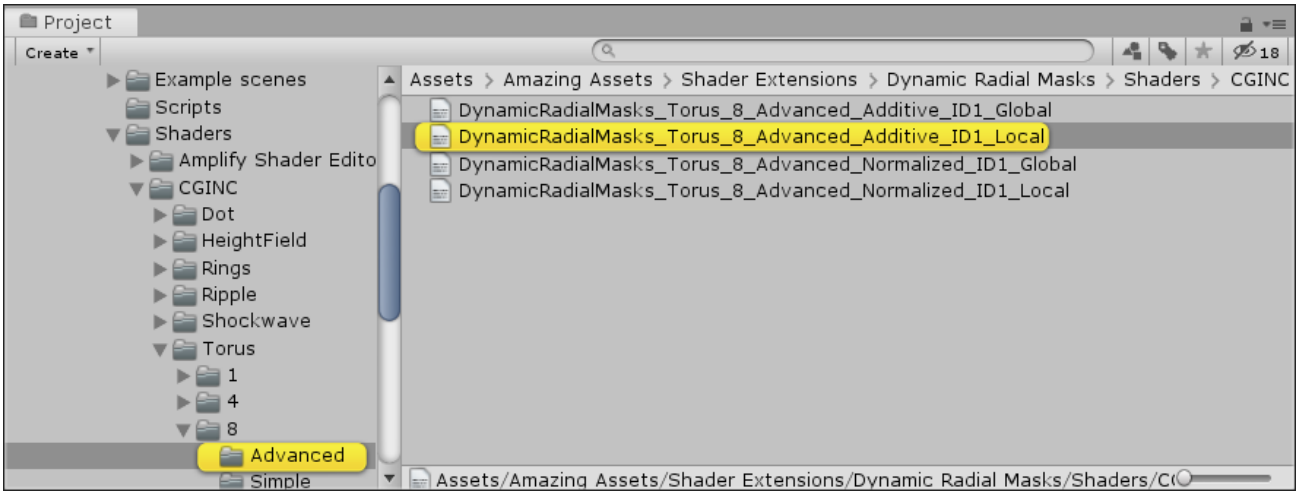
Step 3 – Updating DRM properties from script

DRM visual properties are declared inside ***cginc*** file in array format. Unity has no API for material editor to display arrays and the only way is to update them from a script.

- To update shader variable we have to know its name.
- Open DRM editor window and make sure settings are the same as on the image below.



- Press on **CGINC** button.
  Unity will navigate Project window to the folder where DRM ***cginc*** file is and highlights it.



- Open file and check first lines of it where class variables are declared.

```
float4 DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local_Position [];
float  DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local_Radius[];
float  DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local_Intensity[];
float  DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local_NoiseStrength[];
float  DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local_EdgeSize[];
float  DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local_Smooth[];
```

Note, variable names follow the same naming rule as the DRM method and ***cginc*** file, described in more details in Dynamic Radial Masks chapter.

- Those variable are visual properties of a mask. Inside script we have to create similar arrays, fill them with required data and then send them to the material using Unity API Material.SetFloatArray and Material.SetVectorArray methods.
- To make working with DRM easy, package includes several example scripts.
  Tutorial scene includes **DRMController** game object and the script with the same name attached.
  DRMController is used to update DRM visual properties of a material from script.

  If inspect **DRMController** script file, on lines 46-54 are declared mask variables in array format.

```
public Vector4[] shaderData_Position;
public float[] shaderData_Radius;
public float[] shaderData_Intensity;
public float[] shaderData_NoiseStrength;
public float[] shaderData_EdgeSize;
public float[] shaderData_RingCount;
public float[] shaderData_Phase;
public float[] shaderData_Frequency;
public float[] shaderData_Smooth;
```

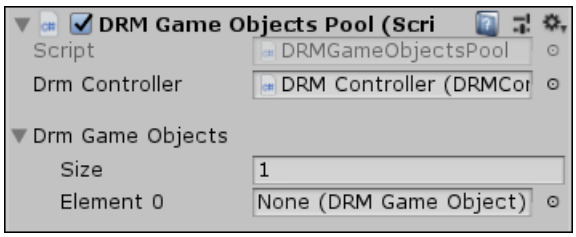
  Those variables are sent to a material using **UpdateShaderData** method (line 152).

  Note, **DRMController** script is explained in more details in Example Scripts chapter.

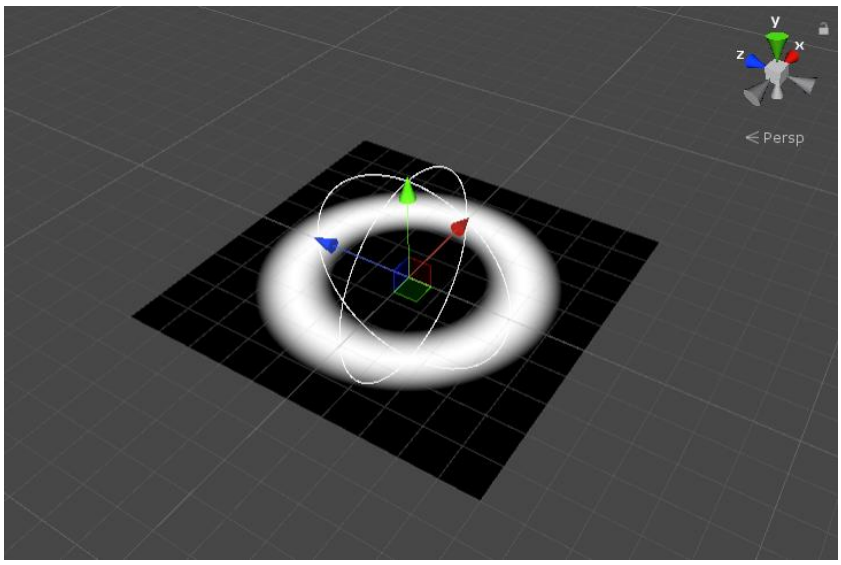
  Our task is to write required data in those variables and **DRMController** will do the rest.

- For this tutorial scene additionally includes two game objects:
  1. **DRMGameObject** represents mask visual properties.
  2. **DRMGameObjectsPool** collects data from **DRMGameObjects** and sends them to the **DRMController** script.

  Currently elements array of **DRMGameObjectsPool** is empty. Drag and drop **DRMGameObject (1)** object here.

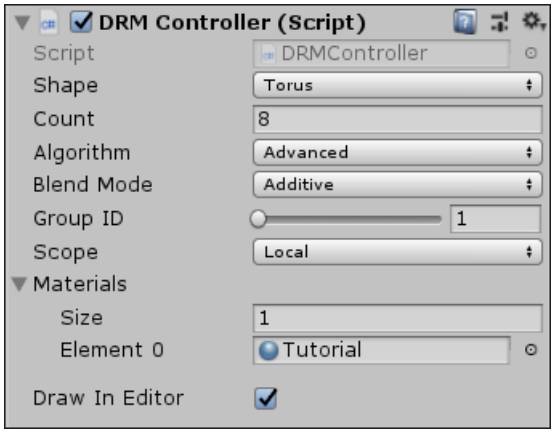


- Adjust properties in **DRMGameObject** or move game object. Material will begin mask rendering.



If mask is not rendered make sure:

1. Plane game object's material is using this tutorial shader.
2. **DRMController** script parameters are the same as on the image below and Materials array is using Plane game object's material.



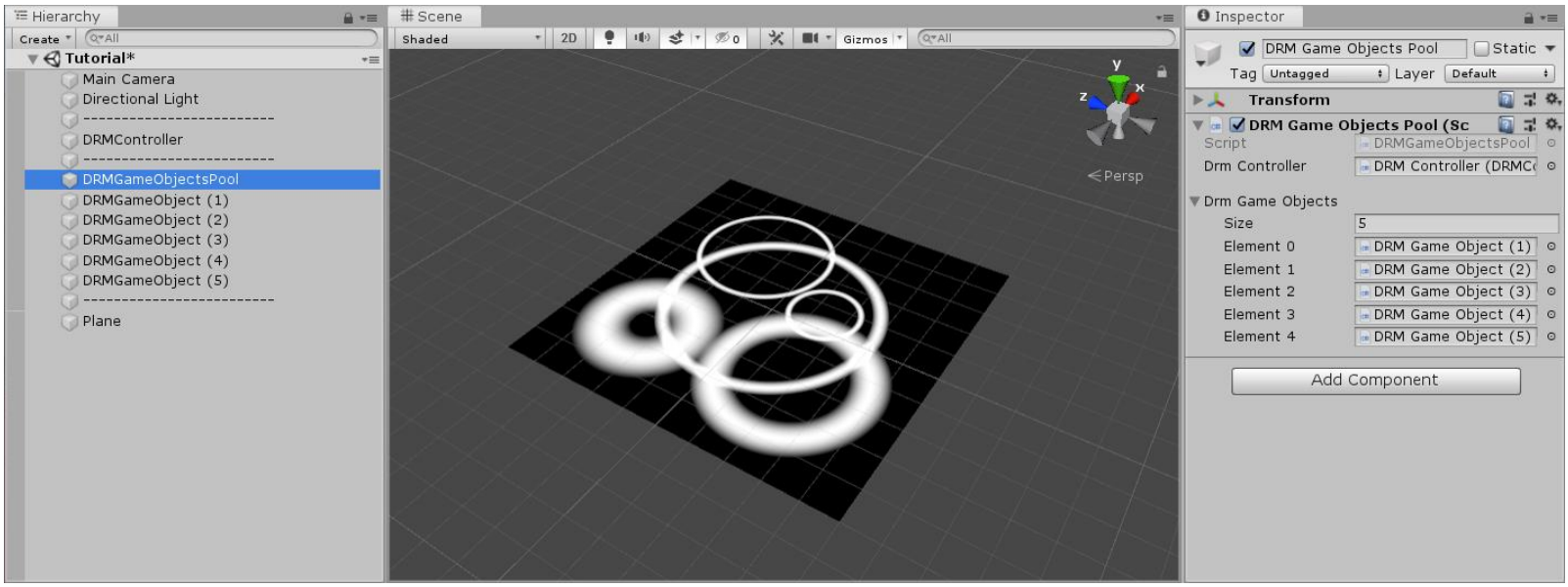
Note, variables are the same as we used in DRM editor window when creating ***cginc*** file.

3. **DRMGameObjectsPool** has assigned **DRMController**.
4. **DRMGameObject** is in elements array of **DRMGameObjectsPool.**
5. **Radius, Intensity** and **Edge Size** properties inside **DRMGameObject** are not 0.

Now mask should be rendered and changing **DRMGameObjects** position and its properties will affect rendered mask.

Note, in this tutorial we have created Torus shape mask, not all properties from **DRMGameObject** are available in this DRM class.

To go a little bit farther, duplicate **DRMGameObject** in the scene and add all of them to the **DRMGameObjectsPool** elements array. Adjust their positions and parameters.



Note, in this tutorial we have created DRM class that can render 8 masks only. Adding more **DRMGameObjects** to the pool will not have effect.

Package includes additional scripts to speed up working with DRM classes. Open **Tutorial 2** scene. This is exactly same scene as in **Tutorial** with the same shaders, but here DRM properties are animated by included **DRMAnimatableObject**. Enter game mode and mouse click on the Plane mesh.

On Every successful click (raycast) **DRMOnMouseRaycast** script creates self-animatable **DRMGameObjects** and adds them to the **DRMAnimatableObjectsPool**, which works the same way and sends data to **DRMController**.

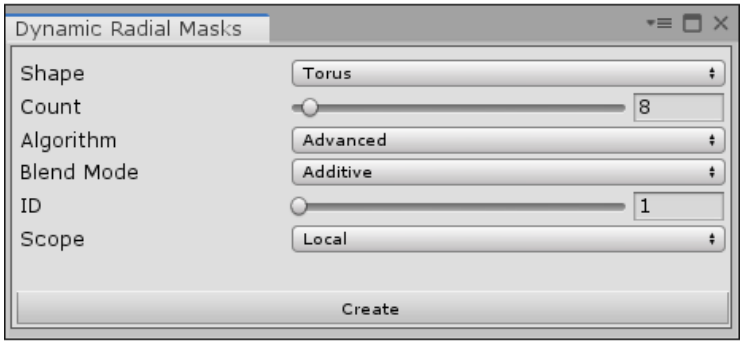
Note, all example scripts are described in Example Scripts chapter and there usage is demonstrated in example scenes.

This is the end of the tutorial. Chapters below describe DRM classes and working with them in more details.

2 Dynamic Radial Masks

Dynamic Radial Masks classes are described in their own ***cginc*** files.
Name of a class, ***cginc*** file and executing method (using in shader) are identical and unique inside a project.

Dynamic Radial Masks class are created inside Unity editor using DRM editor window.



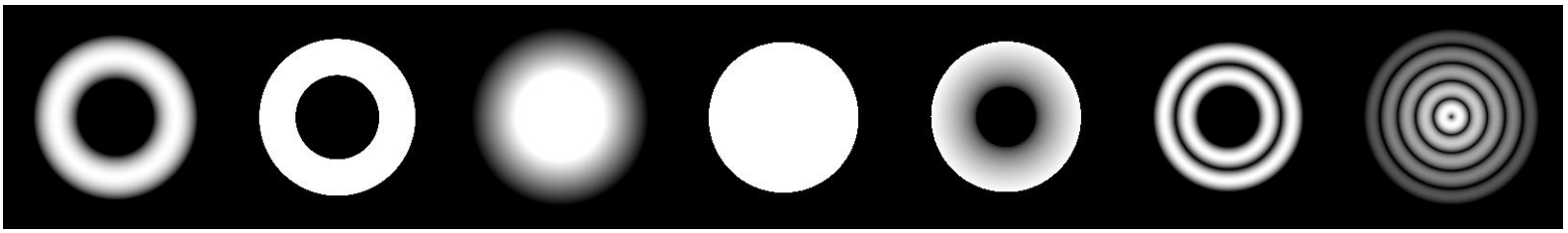
Note, DRM editor window is described in details in Editor Window chapter.

DRM class settings are encoded in its own name. For example in Tutorial chapter we have created **DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local** class and each segment of this name represents class settings.

2.1 Shape

Visual rendering shape of a mask. **Dynamic Radial Masks** can render seven types of spherical projections.

Torus Tube Height Field Dot Shockwave Rings Ripple



Each one uses its own calculation algorithm and has different performance cost in the term of instructions count.
For shape calculation DRM uses several properties (visual characteristics), they are declared inside DRM class ***cginc*** file and are always updated from script.
Chart below displays available properties for each shape.

	Position	Radius	Intensity	Noise Strength	Edge Size	Rings Count	Phase	Frequency	Smooth
Torus	●	●	●	●	●				●
Tube	●	●	●	●	●				
Height Field	●	●	●	●	●				●
Dot	●	●	●	●					
Shockwave	●	●	●	●	●				●
Rings	●	●	●	●	●	●			●
Ripple	●	●	●	●			●	●	●

2.2 Count

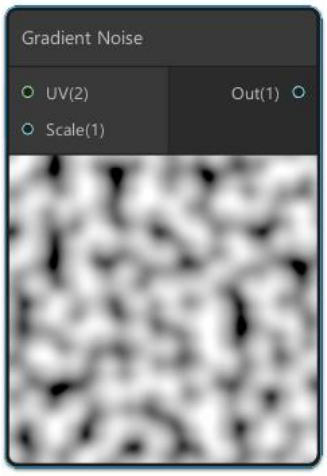
Defines shapes (elements) count DRM class can render using its execution method.
Resultant mask value calculated inside a shader (returned by DRM method) is the result of mixing multiple **Count** of shapes in the ***for loop*** cycle. Performance cost of the DRM class depends on this value. More shapes class can render more expensive it is.

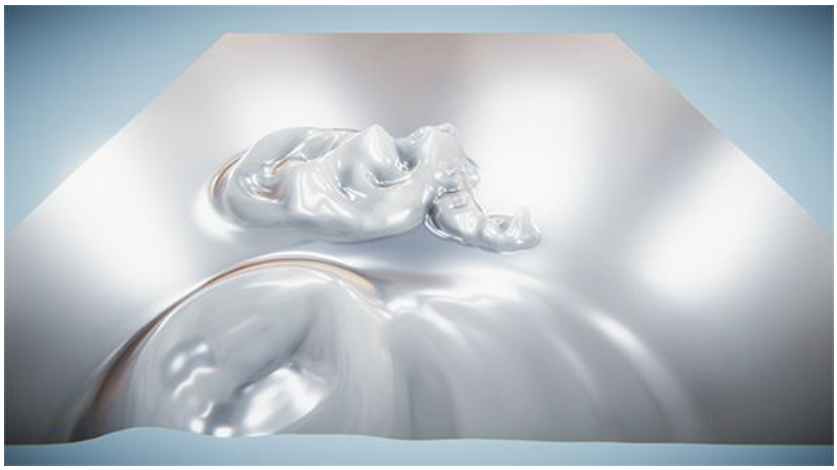
In the Tutorial chapter we have created **DynamicRadialMask_Torus_8_Advanced_Additive_ID1_Local** mask class, and according to its name resultant mask value of this class consisting of **8 Torus** shape elements.

When using DRM inside a shader take note that various GPU's have different limitations on the instructions count per-shader and **Count** parameter must be kept as low as possible. While high-end GPU's can render hundreds of mask elements, same shader may fail to compile on mobile devices.

In the Tutorial chapter we have used mask class with **8 Torus** shape elements. Performance cost for this effect is 8 * 15 = 120 instructions count. Is it high or low? It depends only on required shader effect.

For comparison, **Gradient Noise** node in the Unity Shader Graph requires 133 instructions.
And the "**Liquid Imitation**" shader from the asset Promo video (available in the package as example scene too) uses **128 Torus** shape elements and total instructions count for it is 128 * 15 = 1920.





Another limitation for GPUs is the variables array size.
Inside ***cginc*** file mask visual properties (position, radius, intensity, etc.) are declared in the array format. Size of those arrays is defined by **Count** parameter.

Modern high-end device can work with shaders containing arrays of variables up to 1024 in size, mobile GPUs of course, are much weaker.

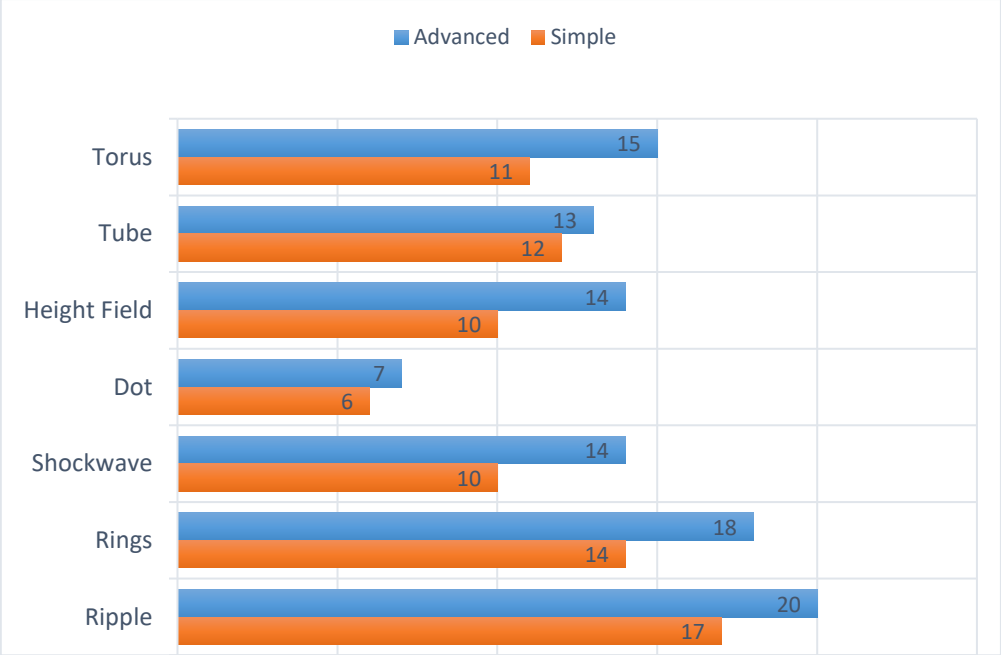
When using DRM inside a shader, always pay attention to those two GPU limitations: Instruction count and variables array size they can work with. Keep **Count** parameter as low as possible.

2.3 Algorithm

Defines algorithm type used for mask calculation:

1. **Advanced** – Uses all visual characteristics of a mask.
2. **Simple** – Optimized version of the **Advanced** method. Uses less instructions and resources. From visual characteristics does not use ***Noise*** and ***Smooth*** features.

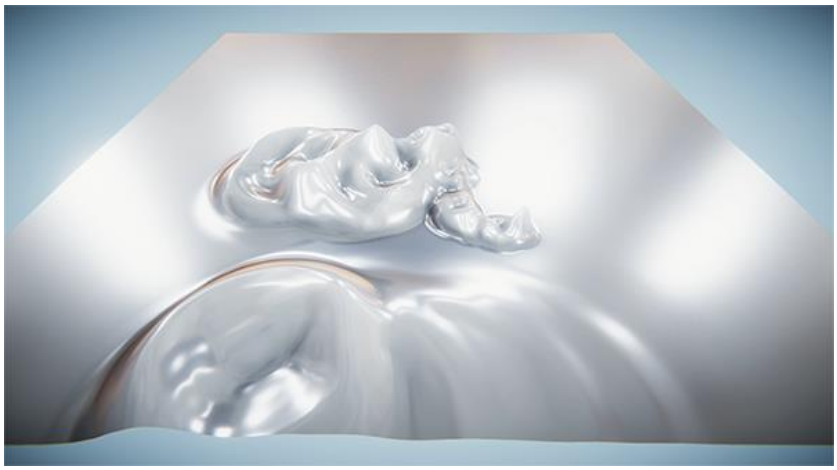
Chart below displays instructions count cost for each shaped mask using **Advanced** and **Simple** algorithms.



2.4 Blend mode

Final resultant mask value inside DRM method is calculated inside ***for loop*** cycle. **Blend Mode** defines how results of each iteration are "mixed".

- **Additive** method simply adds result of one iteration to the previous.
  Resultant value of such mask can be in the range of [0, ∞]. This method is suitable for vertex displace shaders, like as "**Liquid Imitation**" shader.



- **Normalized** method smoothly mixes (without clamp) result of one iteration with previous and guarantees resultant mask value to be smooth and in the range of [0, 1].
  Normalized method uses one instruction more compared to **Additive** blend mode.
  When using **Normalized** method, make sure mask visual characteristic **Intensity** value is in the range of [0, 1].

2.5 ID

Sometimes inside a shader it is required to use several identical mask classes, separately from each other.
For example Bullet Hole shader, where cuts in the mesh depends on the bullet types.
For each bullet type can be used identical mask classes, but with different IDs:
DynamicRadialMask_Dot_8_Advanced_Additive_ID1_Local – for small bullets
DynamicRadialMask_Dot_8_Advanced_Additive_ID2_Local – for rocket
DynamicRadialMask_Dot_8_Advanced_Additive_ID3_Local – for laser

Visual properties of those three masks will be controlled independently from each other using three separate **DRMController** scripts.

Note, instruction count cost of this example shader will be the same as one **Dot_24_Advanced_Additive** mask.

2.6 Scope

Defines scope of the mask properties (visual characteristic variables) inside a shader.

- **Local** variables are updated per-material. Different materials of the same shader may have different values for **Local** variables. Changing such variable in one material does not modify it in other materials.
  For example project may contain any amount of materials using Unity Standard shader and modifying any of the properties in one of them is not reflected on other materials.

- **Global** variables are shared by all materials and changing them in one place is instantly visible to all other materials which are using that variables.

Good example of **Global** variables is the Sonar effect.

In this example scene mask properties creating Sonar effect are same for all materials, so why update those properties for each material individually, when it is possible to update them just once and all materials will instantly receive those changes.

3 Shader Implementation

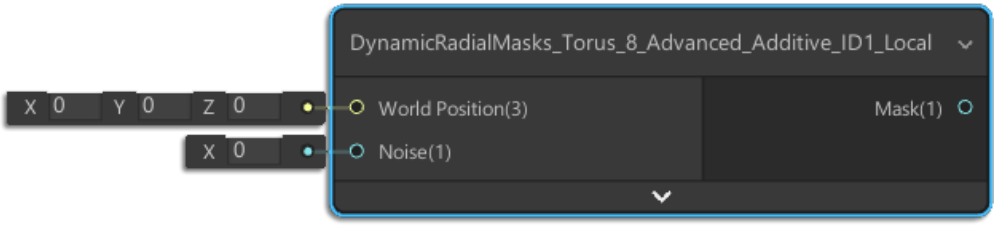
In Tutorial chapter is described process of implementation DRM into hand written shaders and shader graphs.

DRM editor window helps to navigate to the required ***cginc*** and custom node files for drag and drop into shader editors and speeding up shader building process.

DRM methods called in shaders always require vertex **world position**. Without it mask calculation will not be correct.

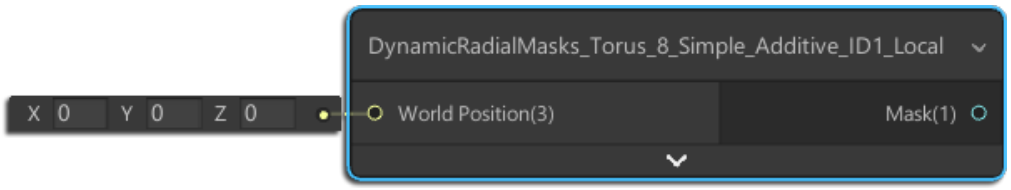
Noise is optional and it can be read from textures or generated procedurally.
Noise Strength variable updated from script is the scale of this shader value.

Noise shader option is available only with mask classes using **Advanced** algorithm.




```
float DynamicRadialMasks_Torus_8_Advanced_Additive_ID1_Local(float3 position, float noise)
```

Simple algorithm does not use **Noise**.

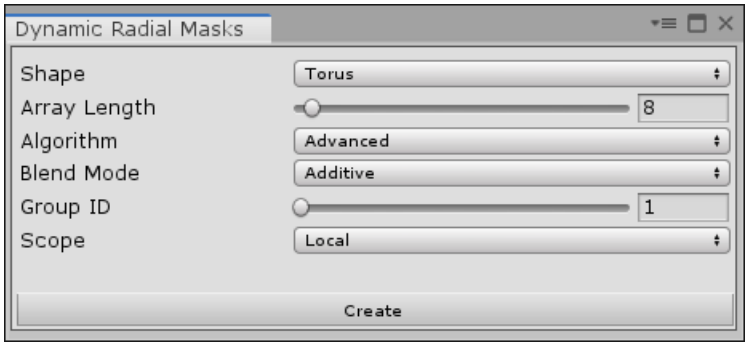



```
float DynamicRadialMasks_Torus_8_Simple_Additive_ID1_Local(float3 position)
```

4 Editor Window

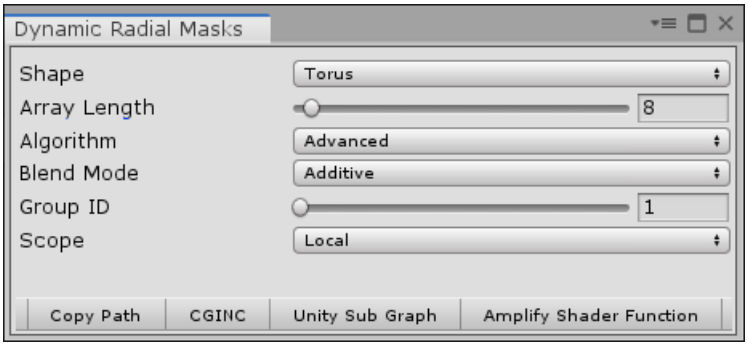
DRM editor window is used to create DRM class ***cginc*** files and helps during shader building.
Editor window can be opened from:

- Menu -> Window -> Amazing Assets -> Dynamic Radial Masks.
- Or, right click inside Project window and from context menu choose Create -> Shader -> Dynamic Radial Masks.



Note, Settings of the DRM editor window are described in Dynamic Radial Masks chapter.

Create button creates DRM class ***cginc*** file and additional node files for **Unity Shader Graph** and **Amplify Shader Editor**. Files are saved inside **Amazing Assets\Dynamic Radial Masks\Shaders** folder, each one in its own sub folder.
Before creating DRM class and files associated with it, editor window checks if such class already exists and replaces **Create** button with:



1. **Copy Path** – Copies ***cginc*** file path to the keyboard memory.
2. **CGINC** – Navigates Unity Project window to the ***cginc*** file and highlights it.
3. **Unity Sub Graph** – Navigates Unity Project window to the Unity Sub Graph file and highlights it. This file can be drag and dropped into the Unity Shader Graph window as a custom node.
4. **Amplify Shader Function** - Navigates Unity Project window to the Amplify Shader Function file and highlights it. This file can be drag and dropped into the Amplify Shader Editor window as a custom node.

5 Houston, we have a problem

Here are suggestions to follow if there is no DRM calculation inside a shader and it is thought it should be there.

Make sure **DRMController** script updating material receives required data. Enable **Draw In Editor** option. It helps to visualize mask objects inside editor.
If nothing is drawn in Editor's Scene view it means that there is no data sending to the material and scripts updating **DRMController** should be checked.

If **DRMController** controller receives mask visual property data, make sure scripts **settings** are the same as DRM class used in a shader.

If shader uses several DRM classes, update them separately – use one **DRMController** for each one.

If DRM class properties are updated locally, make sure target material is in the **Materials** array of the **DRMController** script.

6 Calculate Mask Using Script

DRM mask effect created inside a shader can be calculated using script too.
It can be used for various effects which does not use shaders, but need interaction with DRM.

For example, check **Example_11_Script_Liquid** scene where sphere objects position (vertical displace) is effected by mask.
Another example is **Example_12_Script_Forest** scene. Tree objects there are animated when they interact with mask controlled by **DRMGameObject**.

DRM script side calculation is available from the **DRMController** script by using **GetMaskValue** method:
`public float GetMaskValue(Vector3 position)` – method takes points world space position and its interaction value with DRM class is calculated based on settings and mask visual properties that **DRMController** currently works with.

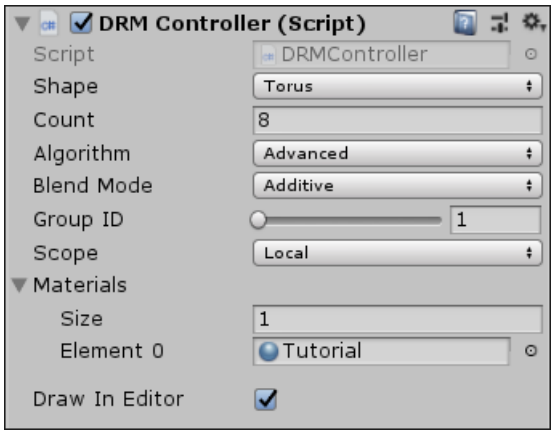
Note, **GetMaskValue** method does not use **Noise** property of a mask.

Value retuned by **GetMaskValue** method is the same as calculated inside shaders.

7 Example Scripts

Package includes several ready to use example scripts making working process with DRM very easy. Using those scripts is not mandatory and they can be replaced with any other custom scripts or be modified for user needs.

7.1 DRMController script



This script is used for updating mask variables inside materials. Those variables are visual properties of a mask and are described in the mask class ***cginc*** file in array format. Unity cannot display arrays inside material editor and the only way left is – updating them from a script.
DRMController script calculates DRM class property names based on provided settings.

Note, they are the same as in the DRM editor window.

Variables in array format are described on lines: 45-54.

```
public Vector4[] shaderData_Position;
public float[] shaderData_Radius;
public float[] shaderData_Intensity;
public float[] shaderData_NoiseStrength;
public float[] shaderData_EdgeSize;
public float[] shaderData_RingCount;
public float[] shaderData_Phase;
public float[] shaderData_Frequency;
public float[] shaderData_Smooth;
```

Those variables are sent to a material using **UpdateShaderData** method (line 152)

Note, purpose of the **DRMController** script is not modifying or somehow adjusting data inside variables declared on line 45-54, it just sends them to a material. What kind of data will be written there and how it is modified is the task of other scripts.

DRMController script works only with one DRM class at a time. Materials expecting data should be in the **Materials** array.
If shader uses several DRM classes, for each one of them must be used separate **DRMController** script, and thus one material can be updated using multiple **DRMController** scripts.

Draw In Editor option is used to visualize **DRMController** script data inside Editor's Scene view.

7.2 DRMGameObject script

Simple class representing mask visual properties. Purpose of this script is to manually control mask properties inside editor or adjust them from other scripts.
Script does not do any calculations and is designed to be used with **DRMGameObjectsPool** script.

Note, even script contains all kind of mask visual properties, their usage depends on mask shape they update in shader and mask algorithm. For example, Simple algorithm does not use Noise and Smooth.

7.3 DRMGameObjectsPool script

This script is used to collect data from **DRMGameObjects** and send them in array format to the **DRMController** script.

7.4 DRMAnimatableObject class

This class is similar to **DRMGameObject**, but with self-animatable properties. This class is designed to be used in **DRMAnimatableGameObjectsPool** script. As **DRMGameObject** is a class and not a script, it can be a member of any script and on required event being duplicated into a pool, where its properties are animated.

7.5 DRMAnimatableObjectsPool script

This script is used to collect data from **DRMAnimatableObjects** and send them in array format to the **DRMController** script.

7.6 DRMOnCollision script

Generates **DRMAnimatableObject** on collision event with game object and adds it to the **DRMAnimatableObjectsPool**.

7.7 DRMOnMouseRaycast script

Generates **DRMAnimatableObject** on successful mouse click (raycast) on the game object and adds it the **DRMAnimatableObjectsPool**.

7.8 DRMOnParticleCollision script

Generates **DRMAnimatableObject** on the particle collision with game object adds it to the **DRMAnimatableObjectsPool**. Script must be assigned to a ParticleSystem.

7.9 DRMOnImpulse script

Generates **DRMAnimatableObject** over time and adds them the **DRMAnimatableObjectsPool**.

8 Example Scenes

This chapter describes example scenes included in the package.

Example scenes are built using **Standard** render pipeline, if project uses **Scriptable** render pipeline (LWRP, URP or HDRP) temporarily disable it from Unity Graphics Settings. Also make sure project uses Unity Post Processing, it can be downloaded from the Package Manager.

8.1 Example_1_All_Masks_Preview

Simple preview of all DRM shapes. Each object uses different material with different mask shape. Mask properties are updated with individual set of **DRMController**, **DRMGameObjectsPool** and **DRMGameObject** scripts.

Because of using **DRMGameObject**, mask properties can be adjusted directly in editor without entering Game mode. Materials are using texture for DRM noise and it can be adjusted using **Noise** property of **DRMGameObject**.

8.2 Example_2_OnMouseRaycast

Enter Game mode and using mouse click on the Plane object. Tube shaped masks will be rendered.
On every successful mouse click (raycast) **DRMOnMouseRaycast** script generates **DRMAnimatableObject** and adds it to the **DRMAnimatableObjectsPool.**
DRM class used in Plane material can calculate mask consisting of 64 Tubes.

8.3 Example_3_OnCollision

Enter Game mode and using mouse click on the Plane object. Blue spheres will be shoot in the Plane direction.
Plane object has **DRMOnCollision** script attached, that generates **DRMAnimatableObject** on every OnCollisionEnter event and adds it to the **DRMAnimatableObjectsPool.**
DRM class used in Plane material can calculate mask consisting of 64 Tubes.

8.4 Example_4_OnParticleCollision

Enter Game mode. Particle System has **DRMOnParticleCollision** script attached, that generates **DRMAnimatableObject** and adds it to the **DRMAnimatableObjectsPool** on every OnParticleCollision event.
DRM class used in Plane material can calculate mask consisting of 64 Tubes.

8.5 Example_5_MultipleMasks

This scene demonstrates using several DRM classes within one shader. In this example using of **Shockwave** and **Dot** masks together.
Shockwave properties are controlled by **DRMOnCollision** script attached to the Plane object generating **DRMAnimatableObject** on every OnCollisionEnter event.
Dot properties are controlled by **DRMOnParticleCollision** script attached to the Particle object generating **DRMAnimatableObject** on every OnParticleCollision event.
Parameters of each DRM class is updated with separate **DRMController** script.

8.6 Example_6_Liquid (Tessellation)

Enter Game mode and hold/move mouse clicked over the Plane object.

In this example shader mask is calculated in vertex shader and its value is used to displace vertices. Additionally is calculated displaced ***normal*** vector for better interaction with lighting in fragment shader.

Note, **DRMOnMouseRaycast** generates 30 **DRMAnimatableObjects** per-second here.

8.7 Example_7_BulletHoles

Enter Game mode and mouse click on the Door. In this example mask value inside shader is used to discard fragments, the same way as Alpha Cutout shaders do.

Note, DRM class used in this shader calculates mask consisting of 64 elements.

8.8 Example_8_ForceField

Enter Game mode and mouse click on the ForceField object. On every mouse click **DRMOnMouseRaycast** creates self-animatable **DRMAnimatableObjects** and adds it to a pool. Demonstration of this example scene is fragmented displace effect created in ForceField shader. Tanique used here is common - vertex displace along its normal vector. Most of vertex displace shaders do this.

For creating fragmented displace effect, mesh is fragmented first. ForceField game object has attached **MeshFragmenter** script that calculates triangles center and its normal vector. Each 3 vertices of a triangle stores this data in UV3 (center position) and UV4 (normal direction) buffers.
Inside shader DRM method requires vertex position to calculate mask and ForceField shader uses triangles center position, not vertex position. Because of this 3 vertexes creating triangle receive similar mask value.

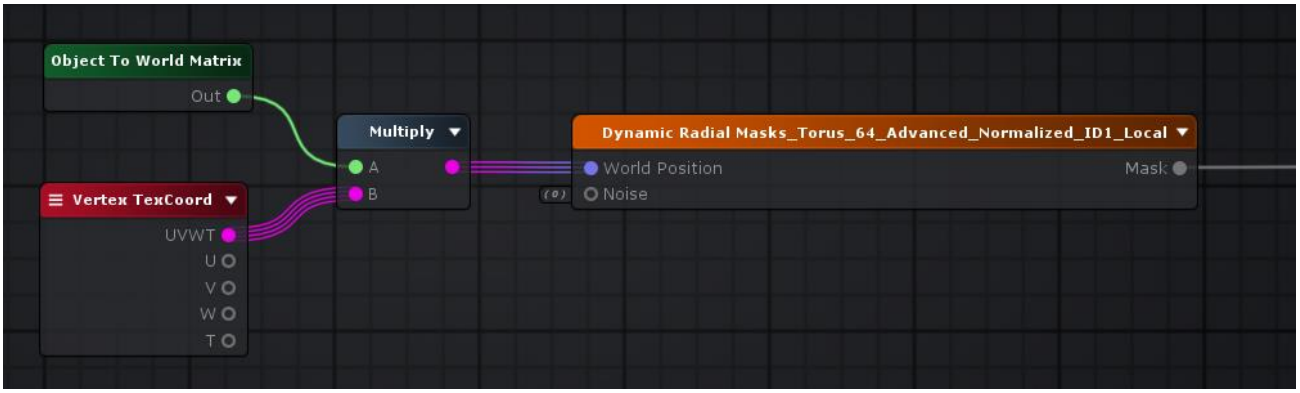


Image above demonstrates reading triangle center position stored inside vertex coordinate (uv3) and multiplying it with world matrix, for converting it into world coordinate system. For this position then is calculated mask value.

Similar way is read triangle's normal (stored in uv4) used for fragment displace direction.

8.9 Example_9_Forest

In this scene mask is controlled using **DRMGameObjects**. All materials are using the same global DRM class and **DRMController** Scope property is set to Global.

Shaders used here simple interpolate property values from "summer" to "winter" based on mask value. DRM properties can be adjusted directly in editor.

Scene contains Particle System, using the same global DRM class in its shader, that effects transparency of the particles. Enter Game mode to see Particle System simulation.

8.10 Example_10_Sonar

This is the same scene as **Example_9_Forest**, but with one more global DRM class used by shaders. This last one is used for creating Sonar effect.

For Sonar effect is used Torus shaped DRM class that is created in constant time interval by **DRMOnImpulse** script.

Scene uses two separate **DRMController** scripts, one for updating "snow" mask, anther for "sonar".

8.11 Example_11_Script_Liquid

This is the same scene as **Example_6_Liquid (Tessellation)**, but with the demonstration of the script side mask calculation. Sphere objects here have simple script attached that retrieves DRM value from **DRMController** script and based on that value adjust objects position.

Note, **GetMaskValue** method used here is described in Caluclation Mask Using Script chapter.

8.12 Example_12_Script_Forest

Another example for demonstration script side DRM calculation. Trees retrieve mask value from **DRMController** and based on its value trigger animation.

Enter Game mode and adjust **DRMGameObject's** position and radius property. Tree meshes on the edge of the mask will start playing animation.

8.13 Example_13_Dissolve

Scene contains two groups of meshes. Meshes in the first group are alwayes hidden in the same position where meshes from another group are visible and vice versa. This is achived by inverting mask value inside shader for the one group of the meshes (check **Invert Dissolve** optoin inside material editor).

Without entering Game mode, change **DRMGameObject** script Radius, Noise Strength and Edge Size properties.