FERM Toolkit for Unity

Warnici
`http://wardddev.com`
info@WARddDev.com
FERM@LapseOfSanity.net

Thank you for purchasing the Fast & Easy RayMarching Toolkit for Unity! In this document we will describe FERM and its functionality in great detail, but first a short summary.

Abstract

The FERM toolkit is a package to be used in the Unity editor. It is an easy to use plugin that allows you to quickly and easily implement raymarching scenes in your project. Raymarching is a rendering technique vastly different from rasterization, which is the default method used in Unity. In raymarching, a scene is defined by its distance function, a single formula that takes position as input and outputs a distance value.
This representation allows you to transform, mix, recurse, and perform many more visually stunning effects, as long as they can be defined in simple terms mathematically. FERM offers heaps of premade shapes and fractals in both 2D & 3D combined with all of the fundamental mixing and modifying operations which can be combined to create everything you can possibly dream up. There are also tools available to manipulate the raymarched scene, via script, physics, mechanim animations and even reacting to sound input! After all, raymarching only really comes to life when it is animated in real time. Still images do not do this technique justice, so take your time to get familiar with all the options.

Contents

1 How does it work

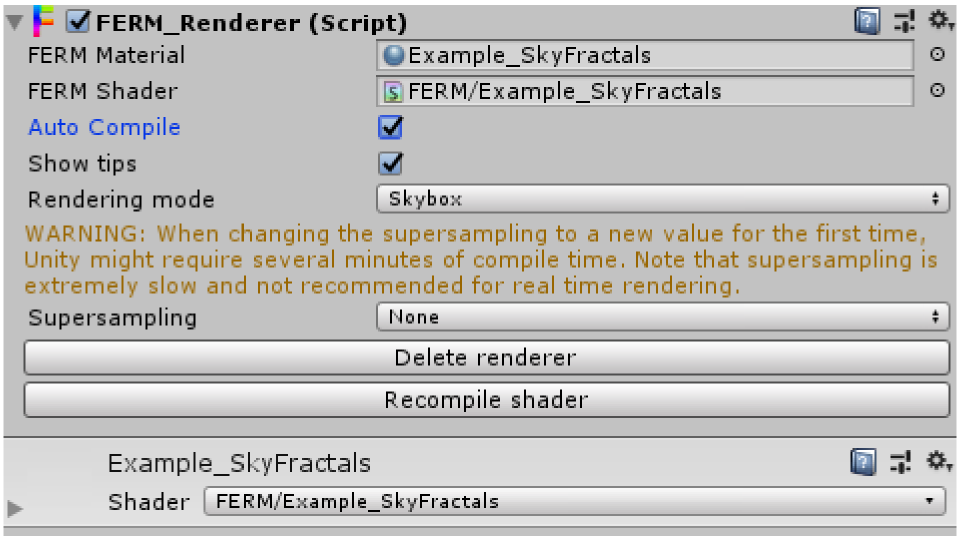
The FERM toolkit implements a raymarching approach, it renders a mathematical representation of a scene using a so-called distance function. A distance functions serves as an estimate (in general a lower bound) for how far away a given point is to a solid surface in the scene. Rendering boils down to 'marching' along every camera pixel ray (similar to ray tracing), taking steps alongs the ray as determined by the distance function. Assuming a proper distance function, this marching will quickly converge, stopping when it gets close enough or diverges. If a surface is found any number of techniques can be used to color a given pixel.

Every FERM Renderer is accompanied by a shader that gets compiled from a shader template and a distance function. These two get combined into a single shader and compiled so that it can run on whatever GPU you are using, achieving real-time performance. Because parameters are baked into the compiled shader-code, extra effort is required to change them without recompiling. Parameters such as transform position and rotation are exposed as parameters in the shader and updated to for example allow moving objects without having to recompile the shader. This same approach allows any number of parameters to be animated on the fly as well. Note that you cannot recompile shaders in an exported Unity game. This means for example that you cannot add new shapes to a scene at run time. In stead, you should have these shapes present, perhaps scaled down or moved very far away and animate them into the scene when desired.

Adding or removing components or changing component types will generally prompt the renderer to recompile. If AutoCompile has been disabled this needs to be done manually each time. Otherwise, each operation will be followed by a short wait time, in which the program recompiles the shader and updates it, showing you the result as soon as possible.

2 Populating a FERM scene

The first thing we will need is a `FERM_Renderer` component, shader and material. These can be created automaticallyby navigating to the GameObject tab at the top of the Unity editor, then going to FERM and create either an unlit or a standard setup (more types still pending). Now you are ready to add visible objects such as primitives and fractals.


FERM_Renderer (Script)
FERM Material: Example_SkyFractals
FERM Shader: FERM/Example_SkyFractals
Auto Compile: ✓
Show tips: ✓
Rendering mode: Skybox
WARNING: When changing the supersampling to a new value for the first time, Unity might require several minutes of compile time. Note that supersampling is extremely slow and not recommended for real time rendering.
Supersampling: None

Example_SkyFractals
Shader: FERM/Example_SkyFractals

Figure 1: The FERM Renderer Component without texture

To create for example an ellipsoid, create a new empty GameObject inside of the FERM Renderer tree and attach the `FERM_Primitive` component. This will select an Arch by default as can be seen in Figure 2. In the background it will update the shader inside of the `FERM_Renderer` to account for the Arch. Now select the ellipsoid and you're done. You can adjust any parameter you'd like and it'll update instantly.

For more complex setups you generally want to combine different shapes with mixers. Note that each gameObject in the Raymarching rendering hierarchy should have either one of the four source shape components or a mixer attached

2.1 Source shapes

There are 4 types of source shapes:

- `FERM_Primitive`
- `FERM_Primitive2D`
- `FERM_Fractal`
- `FERM_Fractal2D`

Each type of each shape defines a single object, primitive shapes like boxes or sphere, or well known fractal shapes like the mandelbulb. These shapes can be set up to have the Unity transform apply to them, so you can move them around in a familiar way. The Unity transform acts exactly like a transform modifier. Options are available to apply this 'modifier' before, after others or not at all.

2.2 Mixers

Mixers apply simple operations to combine different shapes, including Union, Intersection and difference operators. Primitives and fractals must be parented to mixers to be combined. Mixers can also be parented to other mixers.

2.3 Modifiers

each object can have any number of `FERM_modifiers` and a single `FERM_Recurse` component. Modifiers are applied to the source shape or mixer in the order in which they occur as Unity components.

2.4 Recursing

Recursing means to repeat a set of modifiers a number of times. These modifiers will be set to occur in a loop so that the iteration count can be modified on the fly. This component is very powerful, since it allows you to generate new fractals and have these change shape dynamically. Note however that the modifiers targeted for recursing should be clean and fast. Otherwise you can end up with a messy scene and/or poor performance. The Recurse component can be set to target all modifiers, or only those that occur before or after it. At the time of writing you cannot mix different recursion loops, which is why you can only have one recurse component per game object. You can however have different layers of recursion, by adding the recurse component to mixers and their children.

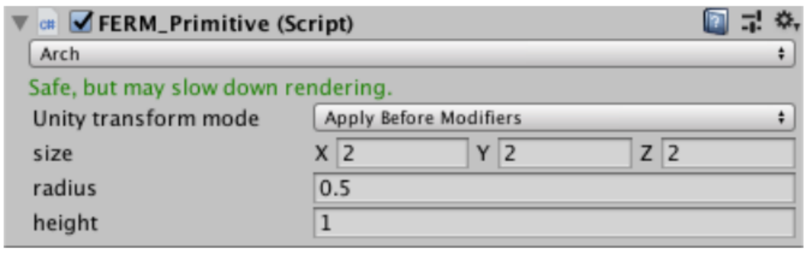
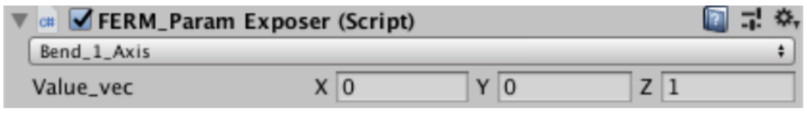

FERM_Primitive (Script)
Arch
Safe, but may slow down rendering.
Unity transform mode: Apply Before Modifiers
size: X 2 Y 2 Z 2
radius: 0.5
height: 1

Figure 2: The default `FERM_Primitive` script

3 FERM animations

3.1 Exposing parameters


FERM_Param Exposer (Script)

Bend_1_Axis

Value_vec X 0 Y 0 Z 1

Figure 3: The Param Exposer Component

As mentioned before, to begin animating a parameter you need to expose it first. To expose a parameter add the Param Exposer component to the relevant FERM object as shown in figure 3. Now you can select the parameter you wish to animate in the list and start animating it as per usual.
New in FERM are multi-expose scripts, such as `FERM_PE5` and `FERM_PE10` which can expose 5 or 10 parameters on the same gameobject. These exposes can also be used besides each-other, although one should take care not to expose any desired parameter multiple times, since this can lead to buggy behaviour.

Setups are also available to have parameters react to sound input. Simply have a AudioSource in your scene that plays music, add the `FERM_ParamAudio` component, target the appropriate parameter and set the music source as input. You can play around with the other parameters to get different effects. In particular the threshold value is very important; it allows the component to detect beats in the music and triggers its response. sample values are provided while the component is receiving music input. its generally best to stick within the range of the smallest and biggest number shown here.

4 FERM Rendering and performance

Once a Raymarching scene has been set up, you might want to take some time to tweak the parameters used for the final render. These can be found on the relevant FERM material. The material used in a `FERM_Renderer` is also shown in the inspector when you select the Game Object with the renderer attached.

4.1 `FERM_Renderer` parameters

Component (type)	**Explanation & examples**
Parameter	Description
Auto Compile	Attempt to immediately update the FERM shader whenever a relevant change occurs in the hierarchy.
Show Tips	Show extra information on each `FERM_Component` to help new users.
Rendering mode	General option that affects depth of the final render. Options are available to render at infinite distance (skybox mode) or render normally with or without writing to the distance buffer. The latter can be used to create very strange, almost glitchy looking effects.
Supersampling	Option that allows multiple raymarch casts to be used per pixel, providing improved anti-aliasing. This function is incredibly slow and is not recommended unless you are trying to record footage in non-real time.

4.2 Colors, textures and UV

A 'standard' FERM shader will try to replicate the standard material of Unity. Note that this material is always affected by global illumination and the main directional light. You can assign a texture and change different color settings in the material. Note that since raymarching has dynamic shapes you cannot do UV unwrapping in the normal sense. Rather you have to use a formula to generate UVs on the fly. Several common options have been provided, for axis aligned unwrapping, spherical and cylindrical maps and a few more special options.

4.3 Performance

A typical full screen raymarching scene will usually not run at 60 fps on an average computer. To ensure good performance you have to spend some time tweaking Raymarching parameters. In general, performance will increase when you lower either of the 4 following rendering parameters.

Parameter	**Description**
Quality Factor	Determines the stopping condition. High quality factors will demand tighter stopping conditions and much more Raymarching steps, affecting performance globally.
Cutoff Factor	This is used to limit the amount of Raymarching steps. Low values will limit rendering to a small number of steps, which can cause holes to appear in the scene. Note also the the ambient occlusion effect depends on the number of steps, so this slider has a typical side effect of darkening/brightening an unlit renderer.
Oversample Factor	Reduces the overall step size, which can patch artefacts in unstable scenes. This factor should always be used as sparingly as possible, since it incurs a lot of performance overhead.
Render Radius	Limits rendering to a spherical region centred on the renderer. This can be used to gain tons of performance when you are rendering a complicated object that is limited in size and only occupies a small section of the camera view. Note that this can also create artefacts if you use it to cut off a sphere from a larger scene. You can more reliably achieve such an effect with the intersection mixer.

4.4 Artefacts

In theory, Raymarching can be done with exact formulas and unlimited iteration, yielding consistently clean scenes. In practice however, it turns outs that if we relax some mathematical conditions we can make huge gains on possible operations and performance.

When building the distance function there are 3 sorts of maths operations you can deal with in terms of reliability, correctness and speed. **Exact** geometries yield exact distance functions as the name implies. They will not yield artefacts unless thoroughly abused. **Bound** geometries yield distance results smaller than the real distance values. They are a bit slower, but don't show any artefacts unless you go crazy with mixing and modifying them. Finally, **Unbound** geometries will change the distance function to a volatile state and must be used carefully to avoid massive artefacts.

You might ask what these artefacts really are, how to recognize them and how to deal with them. This depends a lot on what's happening, but in general you can distinghuish 4 situations.

Artefact type	Description
Holes	These are fairly straight forward, they occur when a ray is being cut off before it can march to the surface and meet the stopping condition. This can be especially annoying since complex shapes will require vastly different amounts of steps to converge. Prevent this by pumping up the cutoff factor as far as needed.
Ghost interactions	Mixing intersections, differences and unions can sometimes lead to objects influencing the scene even though they are invisible otherwise. This problem is baked into your rendering setup and can only be resolved by changing it. Replace differences and intersections with unions where possible. Unions are always the most stable way of mixing shapes.
Overshooting	This can happen when the distance function has been distorted by a powerful modifier. The distance function is no longer exact, allowing some rendering paths to overshoot the stopping condition and land inside an object. You can combat this with oversampling, but mind the performance cost of doing this! Otherwise you can prevent this by tweaking the parameters of the culprit modifier.
Fundamental	The distance function can sometimes be straight up bugged, or just distorted beyond any level of human perception. These tend to play with the camera view in odd ways and can sometimes be surprisingly beautiful. Other times you just get noise. There's no easy way to fix these other than removing whatever caused it in the first place.

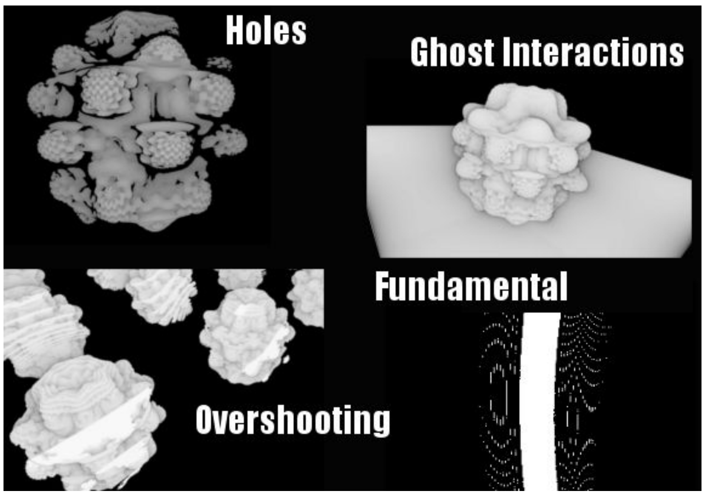


Figure 4: Ferm artefacts

5 Tips & tricks

Some extra info which doesn't fit in any particular chapter.

- Although we talked a lot about avoiding artefacts, they can sometimes be pretty in their own right. Just add a bend modifier to literally anything and crank up the strength to 10000. Enjoy the view.
- The easiest way to make a custom fractal is to start with any primitive, then add a transform and a few mirror modifiers. Scale the object down with the transform, and set up the mirrors (and position) so the final object takes up roughly the same space as the original. Add a recurse modifier and tweak the rotation to make a fractal roll out. You can do this very easily with for example a cube, just scale it by half and add 3 mirrors to restore the original cube. Recurse 20 times, change the rotation and you've got art.

- If you've got an idea of a scene you want to make, turn off the AutoCompile function and compile manually until it's almost finished. Use AutoCompile only to tweak the last bits.

- You can actually modify raymarching parameters via script. This is especially useful for creating actual games using this sort of rendering, in stead of just having some fractal in the background for visual effect only. Material parameters can be accessed in the standard unity way. Raymarching parameters can be accessed with a reference to a `FERM_Component` using the function `SetParam(name, value)`. This can by any component, including primitives, fractals, modifiers, mixers and even the recurse component.

- Since the Raymarching naturally responds to the Unity transform value, you can use FERM in conjunction with Unity physics. Have a few balls mix smoothly with a box and attach colliders matching their shapes. Attach Rigidbodies to the balls and they will roll over the box, yielding moving blob shapes. Unfortunately, you cannot use mesh colliders to automatically apply colliders to fractals (yet). In stead you should add basic colliders, or even use 3D models for mesh colliders where needed.

- The Unity transform can behave very strangely if you apply it before a set of modifiers. This can create very trippy effects in conjunction with standard motion as say performed by the physics engine.

6 Script overview

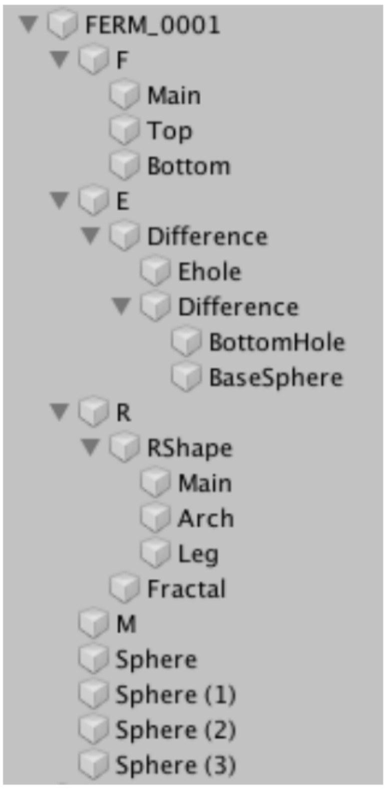

```
FERM_0001
  F
    Main
    Top
    Bottom
  E
    Difference
      Ehole
      Difference
        BottomHole
        BaseSphere
  R
    RShape
      Main
      Arch
      Leg
    Fractal
  M
  Sphere
  Sphere (1)
  Sphere (2)
  Sphere (3)
```

Figure 5: The FERM logo component tree

Component (type)	Explanation & examples
Renderer	Base class for FERM, handles one rendering pass trough a raymarching scene.
Characterizing Component	This is a component class containing all primitives, fractals and the Mixer. Every member of this class defines a distance function. You can only have on of these per game Object. (This base class is abstract and cannot be added directly).
Primitive	This represents a basic shape like a rounded box, ellipsoid, pyramid, etc. We offer 21 different primitives, some infinite, some rounded and some capped.
Primitive 2D	The 2D equivalent of regular Primitives like a triangle, plane, polygon, etc. We offer 7 different primitives in 2D.
Fractal	This represents a recursively defined shape, like a Mandelbulb, Sierpinski tetrahedron or Kock tetrahedron.
Fractal 2D	The more conventionally known 2D fractals like the Mandelbrot & Julia set.
Modifier	This component will modify an existing distance function of a FERM object it is attached to. You can bend, scale, repeat, twist, inflate, .. a FERM object. We offer 16 different modifiers. Modifiers will be applied from top to bottom when there are multiple.
Mixer	This component will merge characterizing components into a single new one. This merging can take the form of an intersection, wobbly union, smooth difference, etc. We offer 7 different mixers.
Recursor	This is a special type of modifier that allows you to repeat the effects of other modifiers.
ParamAccess	Abstract class that facilitates special scripts that allow you to animate raymarching parameters
ParamExposer	Exposes a copy of a parameter that can be affected by mechanim animations. Use this for scriptless animations.
MultiParamAcce	Abstract class that facilitates special scripts that allow you to animate multiple raymarching parameters
PEX	Exposes X parameters that can be affected by mechanim animations.
ParamAudio	Allows a specific parameter to be controlled by sound input. Use this to create audioreactive 'beat' sound scapes.
Caster	Helper script for probing and raycasting the in Raymarching scene via c# scripting.
RaymarchResult	Similar to RaycastHit, entails resulting information about a raymarching raycast.

7 Script casting

Apart from the purely visual effects, FERM also allows the user to probe the raymarching space directly via c# scripting. Casting is facilitated via the `FERM_Caster` script, which can be obtained from the relevant `FERM_renderer` via `GetCaster()`. The probing typically happens in one of 2 fashions, either by doing a raymarching cast into the FERM scene or by probing the distance function directly. Both methods are quite useful for designing games around the raymarching space, though building a valid, artefact-free scene becomes doubly important.

Typical examples of applications include having a character be able to stand on FERM generated terrain by casting distance down, whilst facilitating collisions with obstacles with a handful of distance probes and if needed a normal calculation. Another example would be restraining a flying camera, preventing it going into negative distance areas.

8 Implementation details

In this section we expand upon the inner workings of the FERM toolkit. This part might get expanded upon in the future. If you want to get started modding the FERM Toolkit you can contact the developers with any questions. You can find contact details at the top of the document.

8.1 Rendering

Since Unity uses rasterization, a workaround is required for us to introduce an alternative rendering technique. FERM works in essence by rendering a quad that is stretched over the screen at all times. This quad has the FERM material applied to it and refers to the Raymarching setup defined by the shader.
In other words, this workarounds let Unity do its regular rasterization business, but hijacks rendering to Raymarching in the shader code. General Raymarching code is included in each FERM shader via cginc includes. The cginc code that is used in FERM is located in the `_Shaders/Include` folder.

8.2 Script casting

In order to probe the Raymarching scene directly via c# script, the entire distance function is calculated in parallell in c#, next to the shader code version. This is another testament to the disgusting workarounds I have been reduced to in order to get FERM working.